Cyberscope

Audit Report
OpenLiquidity

June 2024

Network	ETH
Address	0xfa955ec865f51c55e3b6ce02528a6844c2eb9c26
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Unresolved
●	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
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	FRV	Fee Restoration Vulnerability	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MVN	Misleading Variables Naming	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	RRA	Redundant Repeated Approvals	Unresolved
●	RRS	Redundant Require Statement	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved

●	L06	Missing Events Access Control	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Contract Name	OpenLiquidity
Compiler Version	v0.8.26+commit.8a97fa7a
Optimization	200 runs
Explorer	https://etherscan.io/address/0xfa955ec865f51c55e3b6ce02528a6844c2eb9c26
Address	0xfa955ec865f51c55e3b6ce02528a6844c2eb9c26
Network	ETH
Symbol	OpenLi
Decimals	9
Total Supply	200,000,000

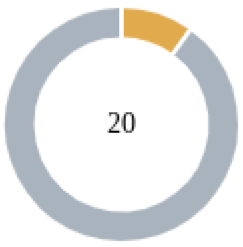
Audit Updates

Initial Audit	04 Feb 2024 https://github.com/cyberscope-io/audits/blob/main/0xlp/v1/audit.pdf
Corrected Phase 2	25 Jun 2024

Source Files

Filename	SHA256
OpenLiquidity.sol	802b94492878d1af63de089c43e515c3f2b578a3d4e8e17049ff93e3d5adb891

Findings Breakdown



🟥	Critical	0
🟧	Medium	2
●	Minor / Informative	18

Severity		Unresolved	Acknowledged	Resolved	Other
🟥	Critical	0	0	0	0
🟧	Medium	2	0	0	0
●	Minor / Informative	18	0	0	0

ST - Stops Transactions

Criticality	Medium
Location	OpenLiquidity.sol#L348,441
Status	Unresolved

Description

Both the `operatorAddress` and the `saleStarterAddress` addresses have the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by setting the `tradingActive` to false. As a result, the contract may operate as a honeypot.

```
if (from != owner() && to != owner()) {
    if (!tradingActive) {
        require(
            from == owner(),
            "Only owner can trade before trading activation"
        );
    }
  ...

    function enableTrading(bool _tradingActive) public {
        require(
            msg.sender == operatorAddress ||
                msg.sender == owner() ||
                msg.sender == saleStarterAddress,
            "Forbidden"
        );
        tradingActive = _tradingActive;
    }
```

Recommendation

It is recommended to include a check within the `enableTrading` function, in order to prevent future updates of the `tradingActive` variable. The team should carefully manage the private keys of the `operatorAddress` , `saleStarterAddress` and owner’s accounts. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

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

Permanent Solution:

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

PTRP - Potential Transfer Revert Propagation

Criticality	Medium
Location	OpenLiquidity.sol#L429,432,437
Status	Unresolved

Description

The contract sends funds to the `_devAddress` and `_daoAddress` contracts as part of the transfer flow. Both of these address are contracts, and as a result, both of them may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
        (bool sentdev, ) = payable(_devAddress).call{value:
primaryAmount}("");
         require(sentdev, "Transfer to dev failed");
         (bool sentdao, ) = payable(_daoAddress).call{value:
secondaryAmount}("");
         require(sentdao, "Transfer to dao failed");

    } else {
         (bool sentdev, ) = payable(_devAddress).call{value:
amount}("");
         require(sentdev, "Transfer to dev failed");
    }
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L424
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 primaryAmount = amount.mul(80).div(100);
uint256 secondaryAmount = amount.mul(20).div(100);
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

FRV - Fee Restoration Vulnerability

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L269,318
Status	Unresolved

Description

The contract demonstrates a potential vulnerability upon removing and restoring the fees. This vulnerability can occur when the fees have been set to zero. During a transaction, if the fees have been set to zero, then both remove fees and restore fees functions will be executed. The remove fees function is executed to temporarily remove the fees, ensuring the sender is not taxed during the transfer. However, the function prematurely returns without setting the variables that hold the previous fee values.

As a result, when the subsequent restore fees function is called after the transfer, it restores the fees to their previous values. However, since the previous fee values were not properly set to zero, there is a risk that the fees will retain their non-zero values from before the fees were removed. This can lead to unintended consequences, potentially causing incorrect fee calculations or unexpected behavior within the contract.

```
   function _tokenTransfer(
        address sender,
        address recipient,
        uint256 amount,
        bool takeFee
    ) private {
        if (!takeFee) remAllFee();
        _transferStandard(sender, recipient, amount);
        if (!takeFee) resAllFee();
    }

    function remAllFee() private {
        if (_taxFee == 0) return;
        _previoustaxFee = _taxFee;
        _taxFee = 0;
    }

    function resAllFee() private {
        _taxFee = _previoustaxFee;
    }
```

Recommendation

The team is advised to modify the remove fees function to ensure that the previous fee values are correctly set to zero, regardless of their initial values. A recommended approach would be to remove the early return when both fees are zero.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L235
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.

```
uniswapV2Pair
```

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.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L182,228,261
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

Specifically, reflection-related variables are assigned to `_tOwned`, which is not intended to be used for reflection.

```
mapping(address => uint256) private _tOwned;

_tOwned[_msgSender()] = _rTotal;

function balanceOf(address account) public view override
returns (uint256) {
    return tokenFromRef(_tOwned[account]);
}
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L413,598
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.

```
uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTok
ens(
    tokenAmount,
    0,
    path,
    address(this),
    block.timestamp
);

    function excludeAccountsFromFees(address[] calldata
accounts, bool excluded)
        public
        onlyOperator
    {
        for (uint256 i = 0; i < accounts.length; i++) {
            _isExcludedFromFee[accounts[i]] = excluded;
        }
    }
```

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.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L409
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
   function swapTokensForEth(uint256 tokenAmount) private
lockTheSwap {
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();
        _approve(address(this), address(uniswapV2Router),
tokenAmount);

uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTok
ens(
            tokenAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
    }
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L412
Status	Unresolved

Description

The contract is designed to approve token transfers during the contract's operation by calling the `_approve` function before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

```
_approve(address(this), address(uniswapV2Router), tokenAmount);
uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTok
ens(
        tokenAmount,
        0,
        path,
        address(this),
        block.timestamp
);
```

Recommendation

It is recommended to optimize the contract by approving the token amount once, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

RRS - Redundant Require Statement

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L97
Status	Unresolved

Description

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

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

Recommendation

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

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	OpenLiquidity.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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L187,188,205
Status	Unresolved

Description

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

```
uint256 private _rTotal = (MAX - (MAX % _tTotal))
uint256 private _tFeeTotal
bool private swapEnabled = true
```

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	OpenLiquidity.sol#L156,178,179,180,184,186,189,190,191,197,198,210,211,212,441
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);
string private constant _name = "OpenLiquidity"
string private constant _symbol = "OpenLi"
uint8 private constant _decimals = 9
mapping(address => bool) public _isExcludedFromFee
uint256 private constant _tTotal = 200_000_000 * 10**_decimals
uint256 public _taxFeeOnBuy = 5
uint256 public _taxFeeOnSell = 5
uint256 public _maxFeeThreshold = 25
address payable public _devAddress
address payable public _daoAddress
uint256 public _maxTxAmount = _tTotal.div(100)
uint256 public _maxWalletSize = _tTotal.div(100)
uint256 public _swapTokensAtAmount = 1000 * 10**_decimals

...
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L188
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
uint256 private _tFeeTotal
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L452
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. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
operatorAddress = operator
```

Recommendation

To avoid this issue, 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.

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L575,587,591,595
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.

```
_taxFeeOnBuy = taxFeeOnBuy
_swapTokensAtAmount = swapTokensAtAmount
_maxTxAmount = maxTxAmount
_maxWalletSize = maxWalletSize
```

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	OpenLiquidity.sol#L566,570
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(
            taxFeeOnBuy >= 0 && taxFeeOnBuy <=
_maxFeeThreshold,
            "Buy tax must be between 0% and _maxFeeThreshold"
        )

require(
            taxFeeOnSell >= 0 && taxFeeOnSell <=
_maxFeeThreshold,
            "Sell tax must be between 0% and _maxFeeThreshold"
        )
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	OpenLiquidity.sol#L436
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 sentdev
```

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	OpenLiquidity.sol#L66,230,231,452,456
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.

```
_owner = msgSender
_devAddress = payable(devAddress)
_daoAddress = payable(daoAddress)
operatorAddress = operator
saleStarterAddress = saleStarter
```

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	OpenLiquidity.sol#L18
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 ^0.8.23;
```

Recommendation

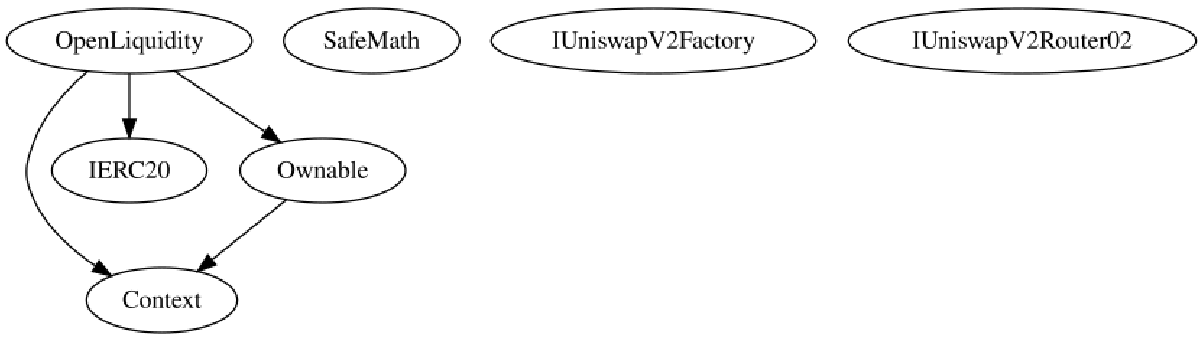
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

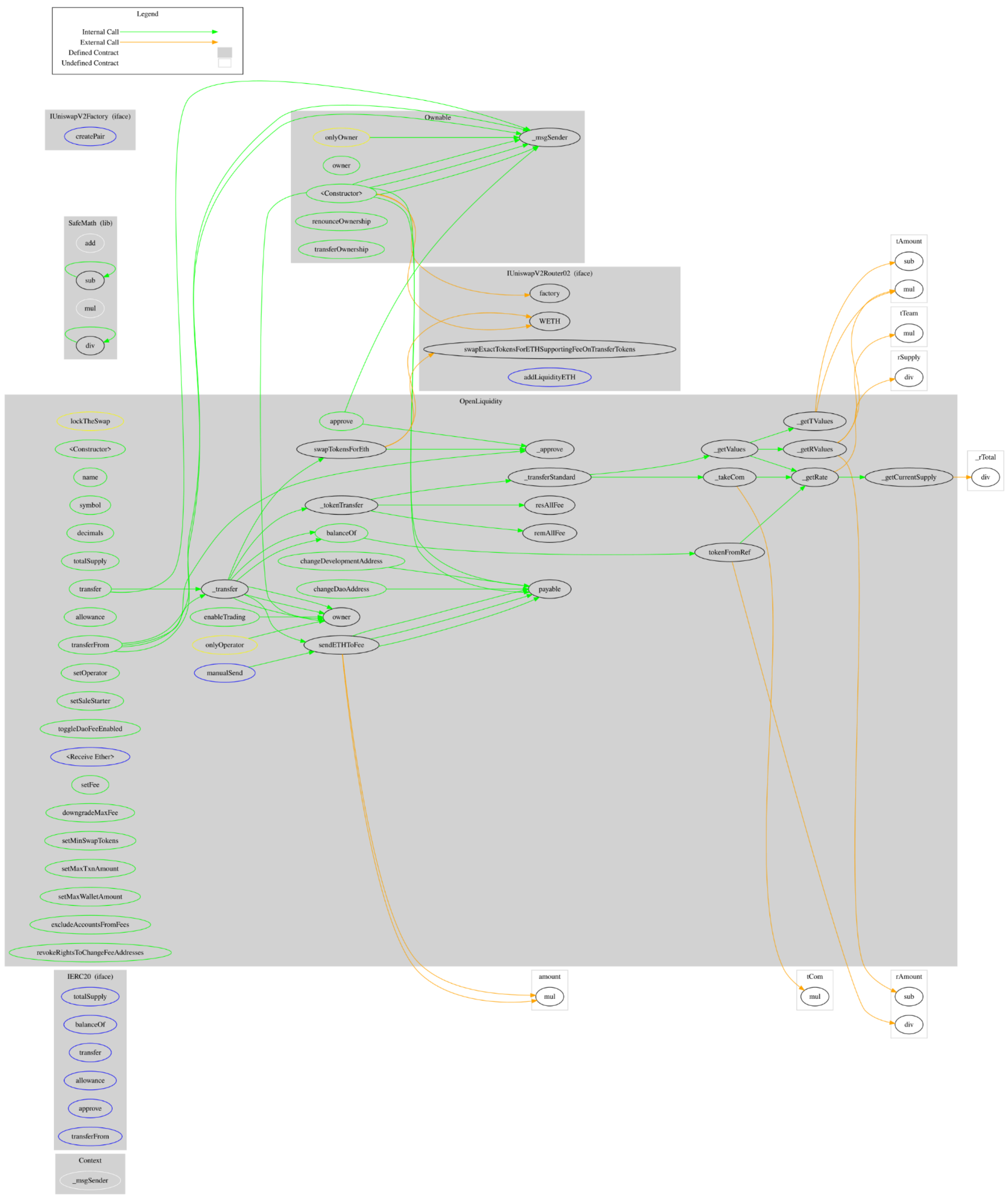
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
OpenLiquidity	Implementation	Context, IERC20, Ownable		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	tokenFromRef	Private		
	remAllFee	Private	✓	
	resAllFee	Private	✓	
	_approve	Private	✓	
	_transfer	Private	✓	
	swapTokensForEth	Private	✓	lockTheSwap
	sendETHToFee	Private	✓	

	enableTrading	Public	✓	-
	setOperator	Public	✓	onlyOwner
	setSaleStarter	Public	✓	onlyOperator
	toggleDaoFeeEnabled	Public	✓	onlyOperator
	manualSend	External	✓	-
	_tokenTransfer	Private	✓	
	_transferStandard	Private	✓	
	_takeCom	Private	✓	
		External	Payable	-
	_getValues	Private		
	_getTValues	Private		
	_getRValues	Private		
	_getRate	Private		
	_getCurrentSupply	Private		
	setFee	Public	✓	onlyOperator
	downgradeMaxFee	Public	✓	onlyOperator
	setMinSwapTokens	Public	✓	onlyOwner
	setMaxTxnAmount	Public	✓	onlyOwner
	setMaxWalletAmount	Public	✓	onlyOwner
	excludeAccountsFromFees	Public	✓	onlyOperator
	changeDevelopmentAddress	Public	✓	onlyOperator
	changeDaoAddress	Public	✓	onlyOperator
	revokeRightsToChangeFeeAddresses	Public	✓	onlyOperator

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IUniswapV2Factory (iface)
createPair
Ownable
onlyOwner
_msgSender
owner
<Constructor>
renounceOwnership
transferOwnership
SafeMath (lib)
add
sub
mul
div
IUniswapV2Router02 (iface)
factory
WETH
swapExactTokensForETHSupportingFeeOnTransferTokens
addLiquidityETH
tAmount
sub
mul
tTeam
mul
rSupply
div
OpenLiquidity
lockTheSwap
<Constructor>
name
symbol
decimals
totalSupply
transfer
allowance
transferFrom
setOperator
setSaleStarter
toggleDaoFeeEnabled
<Receive Ether>
setFee
downgradeMaxFee
setMinSwapTokens
setMaxTxnAmount
setMaxWalletAmount
excludeAccountsFromFees
revokeRightsToChangeFeeAddresses
approve
swapTokensForEth
_approve
_transferStandard
_tokenTransfer
resAllFee
balanceOf
remAllFee
changeDevelopmentAddress
_transfer
changeDaoAddress
payable
enableTrading
owner
onlyOperator
sendETHToFee
manualSend
_getTValues
_getValues
_getRValues
_takeCom
_getRate
_getCurrentSupply
tokenFromRef
_rTotal
div
amount
mul
tCom
mul
rAmount
sub
div
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Context
_msgSender

Summary

OpenLiquidity contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. OpenLiquidity is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. There are some functions that can be abused by the owner like stop transactions. There is also a limit of max 5% fee on buy and sell transactions.

The contract's ownership has been renounced. The information regarding the transaction can be accessed through the following link:

https://etherscan.io/tx/0xaa13a40751b71e8c4e330df808490de38e46ea3746a5aab62fb0f53b9702a590

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