Cyberscope

Audit Report

Leprechaun

June 2024

Network BSC

Address 0xd445f4cCb4c73AF58bBa206bf255918e85c19b65

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	UTFA	Unnecessary Transfer Fee Assignment	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MVLN	Misleading Variable lastTxTime Name	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	RED	Redudant Event Declaration	Unresolved
●	RAC	Redundant Allowance Check	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	UTA	Unnecessary Token Approval	Unresolved
●	UWAS	Unnecessary Winners Array Shifting	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

Table of Contents

Review

Contract Name	Leprechaun
Compiler Version	v0.8.19+commit.7dd6d404
Optimization	No
Explorer	https://bscscan.com/address/0xd445f4ccb4c73af58bba206bf255918e85c19b65
Address	0xd445f4cCb4c73AF58bBa206bf255918e85c19b65
Network	BSC
Symbol	LPC
Decimals	9
Total Supply	1,000,000,000,000,00
Badge Eligibility	Yes

Audit Updates

Initial Audit	09 May 2024 https://github.com/cyberscope-io/audits/blob/main/lpc/v1/audit.pdf
Corrected Phase 2	04 Jun 2024

Source Files

Filename	SHA256
Leprechaun.sol	0dafabe9d4989edda0aefaad36f0dfc881959de7b53f446cae452e4aee854f1f

Findings Breakdown

Critical	0
Medium	0
Minor / Informative	14

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	14	0	0	0

UTFA - Unnecessary Transfer Fee Assignment

Criticality	Minor / Informative
Location	Leprechaun.sol#L530
Status	Unresolved

Description

There exists an unnecessary assignment of the transfer fee (transferfee) to a variable named fee. The contract employs conditional logic to determine the appropriate fee based on whether the transaction is a buy, sell, or transfer operation. However, in cases where neither buying nor selling applies (isbuy and issell are both false), the transfer fee is assigned to the variable fee. Notably, the transfer fee (transferfee) is always defined as zero (0), rendering the assignment redundant.

```
uint256 public constant transferfee = 0;
...
uint256 fee;
if (isbuy) fee = buyfee;
else if (issell) fee = sellfee;
else fee = transferfee;
if (fee == 0) return amount;
```

`

Recommendation

The team is adviced to eliminate the unnecessary assignment of the transfer fee when neither buying nor selling conditions apply. Since the transfer fee is consistently zero, directly return the amount without assigning it to fee. By implementing these recommendations, the contract's efficiency can be improved, and its logic becomes clearer and less prone to misinterpretation.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	Leprechaun.sol#L317,341
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.

```
usdSwapAddress
lpPair
```

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.

MVLN - Misleading Variable lastTxTime Name

Criticality	Minor / Informative
Location	Leprechaun.sol#L501
Status	Unresolved

Description

In the provided contract, there exists a variable named lastTxTime which suggests it represents the timestamp of the last transaction. However, upon closer inspection, it is evident that this variable is only assigned a value within the context of buy transactions (is_buy(from, to)). Consequently, lastTxTime does not accurately reflect the timestamp of the last transaction but rather signifies the timestamp of the last buy transaction. The current naming convention may lead to confusion and misinterpretation, potentially resulting in errors during contract maintenance or debugging.

```
if (is_buy(from, to)) {
    lastTxTime = block.timestamp;
    ...
}
```

Recommendation

Update the variable name lastTxTime to a more descriptive and accurate name, such as lastBuyTime, to reflect its actual purpose. This adjustment will enhance code readability and reduce the likelihood of misunderstanding its intended functionality.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Leprechaun.sol#L398
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 setNoFeeWallet(address account, bool enabled) public
onlyOwner {
    _noFee[account] = enabled;
}
```

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.

RED - Redudant Event Declaration

Criticality	Minor / Informative
Location	Leprechaun.sol#L310,312
Status	Unresolved

Description

The contract uses events that are not emitted within the contract's functions. As a result, these declared events are redundant and serve no purpose within the contract's current implementation.

```
event _changeThreshold(uint256 newThreshold);
event _changeFees(uint256 buy, uint256 sell);
```

Recommendation

To optimize contract performance and efficiency, it is advisable to regularly review and refactor the codebase, removing the unused event declarations. This proactive approach not only streamlines the contract, reducing deployment and execution costs but also enhances readability and maintainability.

RAC - Redundant Allowance Check

Criticality	Minor / Informative
Location	Leprechaun.sol#L555
Status	Unresolved

Description

Within the provided contract, there exists a redundant allowance check within the internalSwapAndLiquify method. This method contains logic to verify the allowance between the contract and a specified address (swapRouter). However, during contract initialization in the constructor, the allowance is explicitly set to the maximum value using `_approve(address(this), address(swapRouter), type(uint256).max)`. Therefore, the subsequent allowance check within internalSwapAndLiquify becomes superfluous, as it always evaluates to true due to the allowance already being set to the maximum value.

```
// constructor
_approve(address(this), address(swapRouter),
type(uint256).max);

// internalSwapAndLiquify method
if (
    _allowances[address(this)][address(swapRouter)] !=
type(uint256).max
) {
    _allowances[address(this)][address(swapRouter)] =
type(uint256).max;
}
```

Recommendation

The team is advised to eliminate the unnecessary allowance check within the internalSwapAndLiquify method. Since the allowance is initialized to the maximum value in the constructor and remains constant throughout contract execution, there is no need to perform redundant checks.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	Leprechaun.sol#L398,435
Status	Unresolved

Description

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

```
function setNoFeeWallet(address account, bool enabled) public
onlyOwner {
    _noFee[account] = enabled;
}

function changeLpPair(address newPair) external onlyOwner {
    isLpPair[newPair] = true;
    emit _changePair(newPair);
}
```

Recommendation

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

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	Leprechaun.sol#L665
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

Specifically, the contract is performing additional calculations by multiplying with 1 ether and then dividing by 1 ether. This introduces unnecessary computational complexity and could potentially lead to precision errors or gas inefficiencies.

```
function minBuyAmountToWin() public view returns(uint256)  {
        uint256 balanceBnb = address(this).balance / (1 ether);
        uint256 minUsd;

        if(balanceBnb >= 10000) { minUsd = 5; }
        else if(balanceBnb >= 1000) { minUsd = 4; }
        else if(balanceBnb >= 100) { minUsd = 3; }
        else if(balanceBnb >= 10) { minUsd = 2; }
        else { minUsd = 1; }

        address[] memory path = new address[](3);
        path[0] = usdSwapAddress;
        path[1] = swapRouter.WETH();
        path[2] = address(this);

        return swapRouter.getAmountsOut(minUsd * (1 ether), path)[2];
    }
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

It is recommended to simplify the calculations by avoiding the unnecessary multiplication and division with 1 ether. This will reduce the computational complexity, minimize the risk of precision errors, and potentially save on gas costs. For example, directly using the balance in wei without converting back and forth can streamline the logic and improve the contract's efficiency.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	Leprechaun.sol#L483,494
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
if (is_sell(from, to) && !inSwap && canSwap(from, to)) {

uint256 amountAfterFee = (takeFee)
    ? takeTaxes(from, is_buy(from, to), is_sell(from, to),
amount)
    : amount;
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

UTA - Unnecessary Token Approval

Criticality	Minor / Informative
Location	Leprechaun.sol#L346
Status	Unresolved

Description

It was identified that the contract's constructor grants an infinite approval to the Uniswap router for both the contract creator and the contract itself. This approval is granted through the `_approve` function with the maximum possible allowance value, `type(uint256).max`, effectvely allowing the Uniswap router to spend tokens from the contract creator without limit.

```
   constructor() {
     ...
    _approve(msg.sender, address(swapRouter),
type(uint256).max);
    _approve(address(this), address(swapRouter),
type(uint256).max);
}
```

Recommendation

It is advised to review the necessity of granting infinite approval to the Uniswap router during contract deployment. If this approval is not essential for the contract's intended functionality, it should be removed or limited to the necessary allowance amount required for specific operations.

By restricting approval to the minimum necessary level, the contract can minimize the potential impact of malicious activities, such as unauthorized token transfers or manipulative trading behaviors, while still maintaining functionality with external protocols like Uniswap.

UWAS - Unnecessary Winners Array Shifting

Criticality	Minor / Informative
Location	Leprechaun.sol#L500
Status	Unresolved

Description

In the provided contract, there exists a process to update a list of winners where new winners are inserted at the beginning of the array while shifting existing winners to accommodate the new entry. However, this approach introduces unnecessary gas costs associated with array shifting operations, particularly as the number of winners increases. Each iteration of the loop to shift winners incurs additional gas expenditure, potentially leading to inefficient contract execution.

```
if (is_buy(from, to) && amount >= minBuyAmountToWin()) {
    lastTxTime = block.timestamp;
    winnersCount = winnersCount < maxWinnersCount
        ? winnersCount + 1
        : maxWinnersCount;
    for (uint8 i = winnersCount - 1; i >= 1; i--) {
        winners[i] = winners[i - 1];
    }
    winners[0] = payable(to);
}
```

Recommendation

Instead of shifting existing winners to make room for new entries at the beginning of the array, consider appending new winners to the end of the array. By adopting this approach, the contract can avoid the gas costs associated with array shifting, as new winners are added without affecting the existing entries.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Leprechaun.sol#L278,279,280
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 potAllocation = 50
uint256 private devAllocation = 25
uint256 private liquidityAllocation = 25
```

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	Leprechaun.sol#L96,283,284,285,303,304,305,306,307,308,309,416,421
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 = "Leprechaun"
string private constant _symbol = "LPC"
uint8 private constant _decimals = 9
event _enableTrading();
event _setPresaleAddress(address account, bool enabled);
event _toggleCanSwapFees(bool enabled);
event _changePair(address newLpPair);
event _changeThreshold(uint256 newThreshold);
event _changeWallets(address newBuy);
event _changeFees(uint256 buy, uint256 sell);

...
```

Recommendation

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

Find more information on the Solidity documentation

https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Leprechaun.sol#L543,582
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 liquidityAllocationHalf = liquidityAllocation / 2
uint256 lpAmount = (newBalance * liquidityAllocationHalf) /
            (potAllocation + devAllocation +
liquidityAllocationHalf)
```

Recommendation

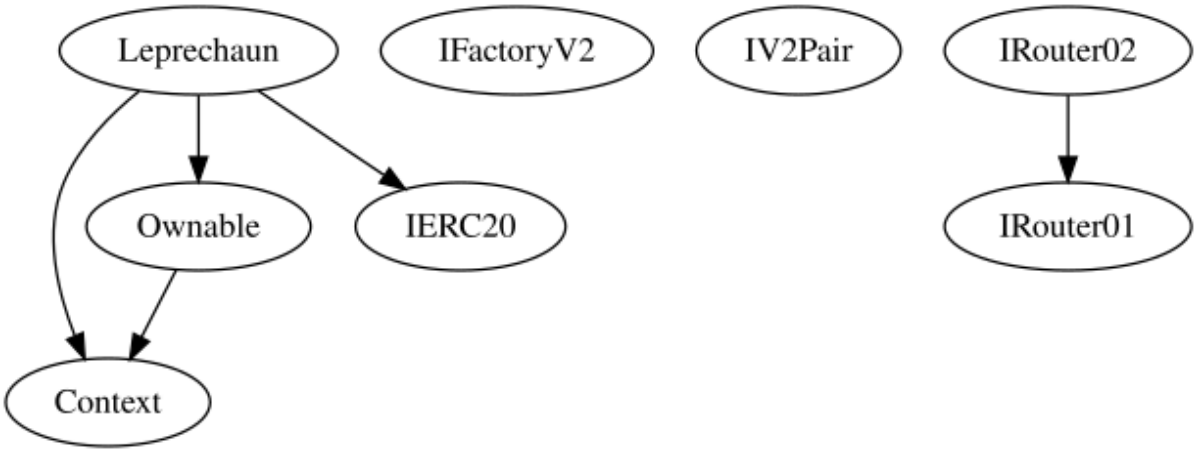
To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

Functions Analysis

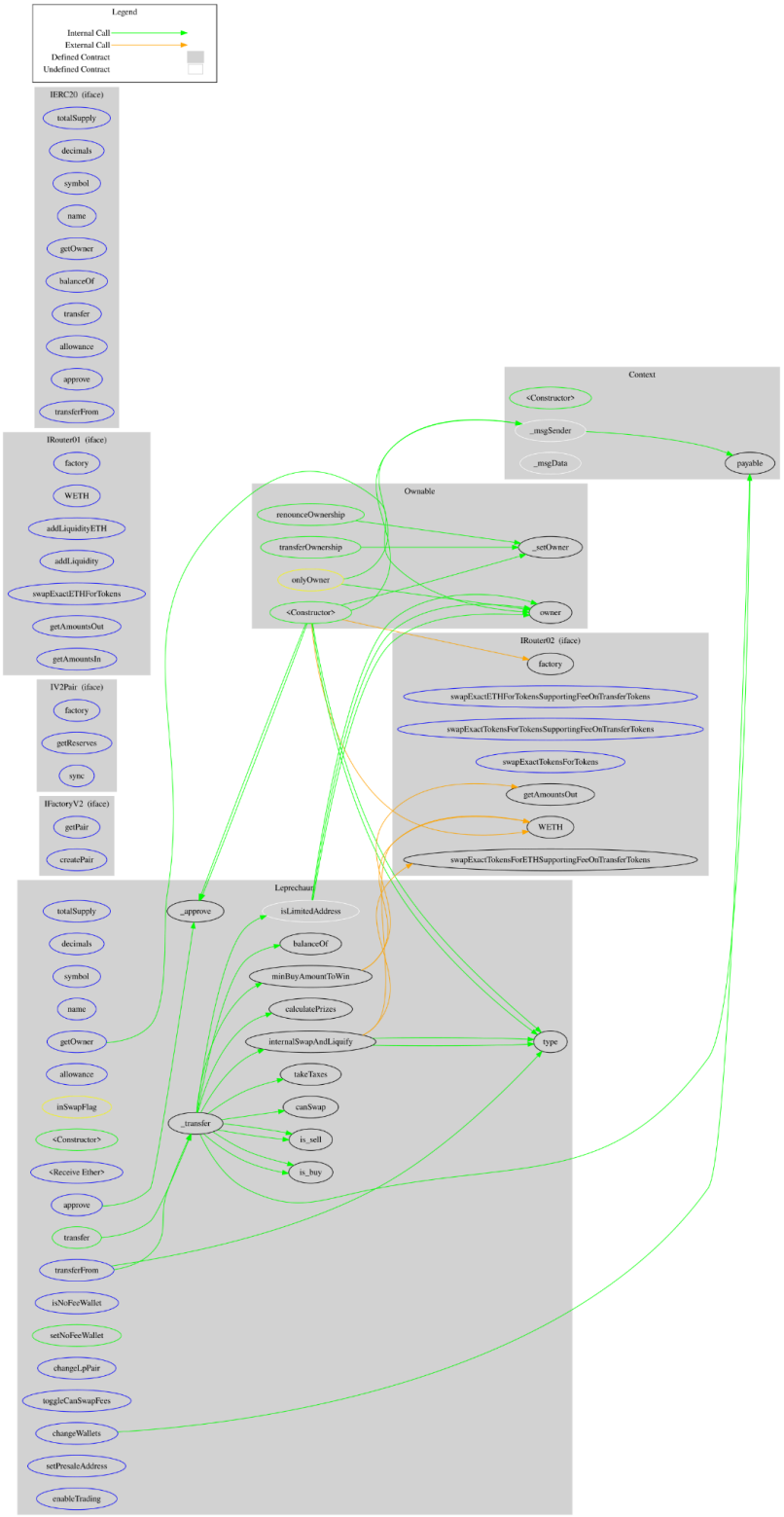
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Leprechaun	Implementation	Context, Ownable, IERC20		
	totalSupply	External		-
	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	allowance	External		-
	balanceOf	Public		-
		Public	✓	-
		External	Payable	-
	transfer	Public	✓	-
	approve	External	✓	-
	_approve	Internal	✓	
	transferFrom	External	✓	-
	isNoFeeWallet	External		-

	setNoFeeWallet	Public	✓	onlyOwner
	isLimitedAddress	Internal		
	is_buy	Internal		
	is_sell	Internal		
	canSwap	Internal		
	changeLpPair	External	✓	onlyOwner
	toggleCanSwapFees	External	✓	onlyOwner
	_transfer	Internal	✓	
	changeWallets	External	✓	onlyOwner
	takeTaxes	Internal	✓	
	internalSwapAndLiquify	Internal	✓	inSwapFlag
	calculatePrizes	Public		-
	minBuyAmountToWin	Public		-
	setPresaleAddress	External	✓	onlyOwner
	enableTrading	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IERC20 (iface)
totalSupply
decimals
symbol
name
getOwner
balanceOf
transfer
allowance
approve
transferFrom
IRouter01 (iface)
factory
WETH
addLiquidityETH
addLiquidity
swapExactETHForTokens
getAmountsOut
getAmountsIn
IV2Pair (iface)
factory
getReserves
sync
IFactoryV2 (iface)
getPair
createPair
Context
<Constructor>
_msgSender
_msgData
payable
Ownable
renounceOwnership
transferOwnership
onlyOwner
<Constructor>
_setOwner
owner
IRouter02 (iface)
factory
swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactTokensForTokens
getAmountsOut
WETH
swapExactTokensForETHSupportingFeeOnTransferTokens
Leprechaun
totalSupply
decimals
symbol
name
getOwner
allowance
inSwapFlag
<Constructor>
<Receive Ether>
approve
transfer
transferFrom
isNoFeeWallet
setNoFeeWallet
changeLpPair
toggleCanSwapFees
changeWallets
setPresaleAddress
enableTrading
_approve
_transfer
isLimitedAddress
balanceOf
minBuyAmountToWin
calculatePrizes
internalSwapAndLiquify
takeTaxes
canSwap
is_sell
is_buy
type

Summary

Leprechaun contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. The fees are set at 8% for buy and sell transactions. The contract's ownership has been renounced. The information regarding the transaction can be accessed through the following link: https://bscscan.com/tx/0xf7bb57604edd93815d4de1d1912e2801cd056820a70640cbc2d50bc56ce43ec3

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