Cyberscope

Audit Report

RAIFI Treasury

June 2025

SHA256

21616ac0ea03e578d3b6fed4efd8994bb5bc64d89aff0a712bc45938984aea0c

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

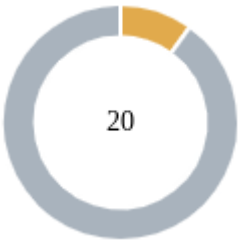
Audit Updates

Initial Audit	12 May 2025
Corrected Phase 2	17 Jun 2025

Source Files

Filename	SHA256
Treasury.sol	21616ac0ea03e578d3b6fed4efd8994bb5bc64d89aff0a712bc45938984aea0c

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	MTF	Missing Transfer functionality	Unresolved
Medium	OCTD	Transfers Contract's Tokens	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	DDP	Decimal Division Precision	Unresolved
Minor / Informative	DBC	Decrease Before Collect	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	IP	Immutable Parameters	Unresolved
Minor / Informative	ISV	Inconsistent State Variables	Unresolved
Minor / Informative	MCM	Misleading Comment Messages	Unresolved
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	UF	Unused Functionality	Unresolved
Minor / Informative	L02	State Variables could be Declared Constant	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

MTF - Missing Transfer functionality

Criticality	Medium
Location	Treasury.sol#L403
Status	Unresolved

Description

The `receiveLPFee` function updates internal accounting variables (`lpFeeBalance` and `totalTreasuryAssets`) based on the input amount, but does not perform any token transfer to enforce this balance update. This results in a discrepancy between the contract's internal state and its actual token holdings.

```
function receiveLPFee(uint256 amount) external {
lpFeeBalance += amount;
totalTreasuryAssets += amount;
}
```

Recommendation

The team is advised to include the necessary transfer logic to ensure that the specified amount is securely transferred into the contract, maintaining consistency between state variables and actual token balances.

OCTD - Transfers Contract's Tokens

Criticality	Medium
Location	Treasury.sol#L517
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `sweepTokens` function.

```
function sweepToken(IERC20 token) external onlyOwner {
uint256 balance = token.balanceOf(address(this));
token.transfer(msg.sender, balance);
}
```

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.

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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Treasury.sol#L311,449,493
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setAuthorizedContracts(address contractAddress, bool isAuthorized)
external onlyOwner {
authorizedContracts[contractAddress] = isAuthorized;
}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	Treasury.sol#L350
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.

```
amount0Desired:RaiReceived,
amount1Desired: amountAction,
```

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.

DBC - Decrease Before Collect

Criticality	Minor / Informative
Location	Treasury.sol#L417
Status	Unresolved

Description

The contract implements the `removeLiquidityV3` function, allowing the owner to decrease the liquidity of a specific position. After reducing liquidity, the function attempts to collect the position's fees. However, if the entire liquidity is removed, it may not be possible to collect the fees accrued prior to the removal.

```
function removeLiquidityV3(uint256 tokenId,uint128 liquidity) external
onlyOwner {
        // 2. Remove liquidity using PositionManager
        positionManager.decreaseLiquidity(
            INonfungiblePositionManager.DecreaseLiquidityParams({
                tokenId: tokenId,
                liquidity: liquidity,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp
            })
        );

        // Sau khi giảm thanh khoản, bạn có thể thu thập (collect)
        // số token còn lại trong vị thế nếu liquidity = 0.
        positionManager.collect(
            INonfungiblePositionManager.CollectParams({ // Thay
NonfungiblePositionManager bằng INonfungiblePositionManager
                tokenId: tokenId,
                recipient: address(this),
                amount0Max: type(uint128).max,
                amount1Max: type(uint128).max
            })
        );
        // 3. Update pool information (không còn trực tiếp như V2)
        // Trong V3, bạn quản lý vị thế thông qua NFT.
        // Thông tin về số dư token bạn nhận được sẽ được cập nhật
        // trực tiếp vào số dư của contract sau khi gọi collect.

        emit LiquidityRemoved(tokenId);
    }
```

Recommendation

The team is advised to perform fee collection before decreasing liquidity to ensure consistency of operations and that all earned fees are properly retrieved.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	Treasury.sol#L295
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.

```
usdtAddress
```

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.

IP - Immutable Parameters

Criticality	Minor / Informative
Location	Treasury.sol#L493
Status	Unresolved

Description

The contract execution relies on parameters that are expected to remain immutable to ensure consistent behavior. However, the owner has the authority to modify these critical parameters via the `setContract` method. This introduces the risk of inconsistent or unexpected behavior if such parameters are altered post-deployment.

```
function setContract( CONTRACTS _contract, address _address ) external
onlyOwner() {
if( _contract == CONTRACTS.USDTTOKEN ) { // 0
usdtToken = IERC20(_address);
}
...
}
```

Recommendation

The team is advised to restrict the ability to modify these parameters, or validate their updates to preserve the contract's integrity and predictability.

ISV - Inconsistent State Variables

Criticality	Minor / Informative
Location	Treasury.sol#L364
Status	Unresolved

Description

The contract maintains state variables that inaccurately represent its actual operations. Specifically, it stores `lpUSDTBalance` and `lpRAIBalance`, which are incremented by `amountUSDT` and `amountRAI`, respectively. However, this does not accurately reflect the liquidity added, as half of `amountUSDT` and the `RaiReceived` variable are used in the liquidity provision.

Recommendation

The team is advised to ensure that the `lpUSDTBalance` and `lpRAIBalance` variables accurately reflect the actual amounts supplied to the liquidity pool. These variables should be updated based on the amounts actually used to avoid discrepancies between the recorded state and the contract's operations.

MCM - Misleading Comment Messages

Criticality	Minor / Informative
Location	Treasury.sol
Status	Unresolved

Description

The contract is using misleading comment messages. These comment messages do not accurately reflect the actual implementation, making it difficult to understand the source code. As a result, the users will not comprehend the source code's actual implementation.

```
// Sau khi giảm thanh khoản, bạn có thể thu thập (collect)
// số token còn lại trong vị thế nếu liquidity = 0.
```

Recommendation

The team is advised to carefully review the comment in order to reflect the actual implementation. To improve code readability, the team should use more specific and descriptive comment messages.

MC - Missing Check

Criticality	Minor / Informative
Location	Treasury.sol#L311
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. Specifically, the contract is not validating that `tickLower` is less than `tickUpper`, the `fee` forms a valid structure and that `token0` and `token1` are the desired addresses.

```
function setLiquidity(int24 _tickLower, int24 _tickUpper,uint24 _fee, address
_token0,address _token1 ) external onlyOwner() {
tickLower=_tickLower;
tickUpper=_tickUpper;
fee=_fee;
token0 =_token0;
token1 =_token1;
}
```

Recommendation

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

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Treasury.sol#L72,157,166,173,214,258
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 setLiquidity(int24 _tickLower, int24 _tickUpper,uint24 _fee, address
_token0,address _token1 ) external onlyOwner() {
tickLower=_tickLower;
tickUpper=_tickUpper;
fee=_fee;
token0 =_token0;
token1 =_token1;
}
```

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.

UF - Unused Functionality

Criticality	Minor / Informative
Location	Treasury.sol#L379,460
Status	Unresolved

Description

The contract includes methods that are not utilized during execution. The presence of such unused functions increases the overall code size and reduces readability.

```
function sqrt(uint256 y) internal pure returns (uint256 z) {
        if (y > 3) {
            z = y;
            uint256 x = y / 2 + 1;
            while (x < z) {
                z = x;
                x = (y / x + x) / 2;
            }
        } else if (y != 0) {
            z = 1;
        }
    }
```

```
function mintRAIForStaking(address recipient, uint256 amount) external
{
...
}
```

Recommendation

The team is advised to review and remove any redundant or unused methods to improve code clarity and maintainability.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Treasury.sol#L250,251,252,253
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 public lpBalance
uint256 public raiBalance
uint256 public daoPoolBalance
uint256 public feeBalance
```

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	Treasury.sol#L75,311,493
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 _mint(address account, uint256 value) external;
int24 _tickUpper
uint24 _fee
address _token1
address _token0
int24 _tickLower
contract == CONTRAC
.USDTTOKEN ) { /
```

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/stable/style-guide.html#naming-conventions.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	Treasury.sol#L86,460
Status	Unresolved

Description

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

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

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

```
function _contextSuffixLength() internal view virtual returns (uint256)
{
        return 0;
    }

...
```

Recommendation

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

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Treasury.sol#L480,481
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.

```
rn raiValue + (usdtValue * (1 ether) / rfvPerRAI);

          return 0;
        }
    }
}
    enum CON
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Treasury.sol#L295,299,300,301,302,303,304,305,307,315,316,499
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.

```
usdtAddress=_usdtTokenAddress
token0 =_raiTokenAddress
bondContractAddress = _bondContractAddress
salesContractAddress = _salesContractAddress
stakingContractAddress = _stakingContractAddress
rewardVestingContractAddress = _rewardVestingContractAddress
contributionValueContractAddress =
_contributionValueContractAddress
devWallet = _devWallet
swapRouterAddress=_swapRouterAddress
token0 =_token0
token1 =_token1
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	Treasury.sol#L2,3
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.20;
pragma abicoder v2;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Treasury.sol#L2
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 ^0.8.20;
```

Recommendation

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

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	Treasury.sol#L396,519
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.

```
usdtToken.transferFrom(msg.sender, address(this), amount)

token.transfer(msg.sender, balance);
```

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.

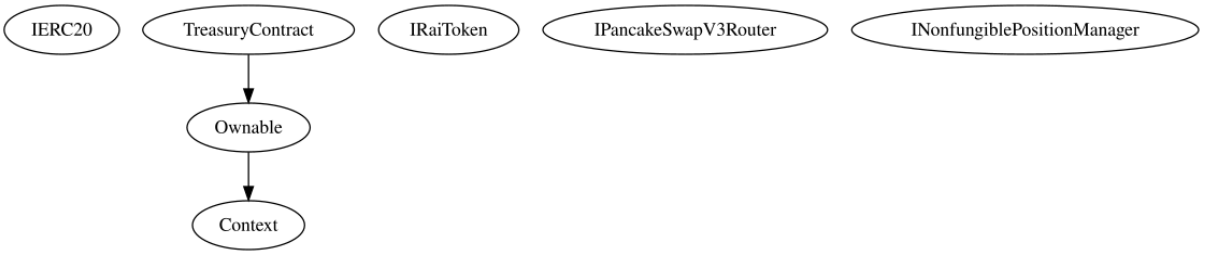
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
	_mint	External	✓	-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner

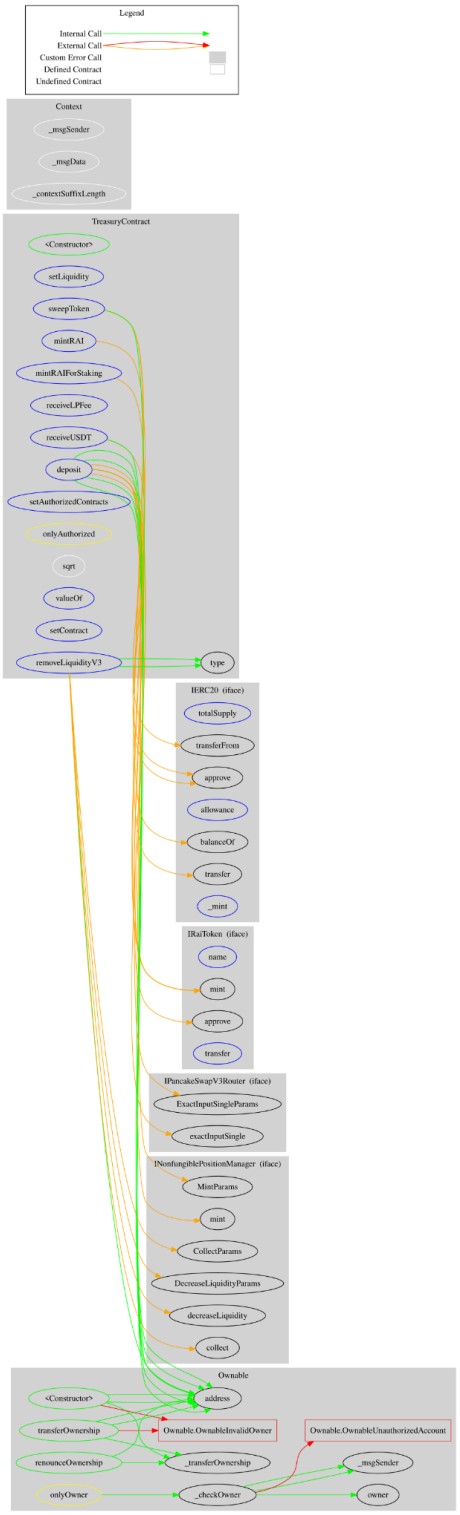
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
IRaiToken	Interface			
	name	External		-
	mint	External	✓	-
	approve	External	✓	-
	transfer	External	✓	-
IPancakeSwapV3Router	Interface			
	exactInputSingle	External	Payable	-
INonfungiblePositionManager	Interface			
	mint	External	Payable	-
	decreaseLiquidity	External	Payable	-
	collect	External	Payable	-
TreasuryContract	Implementation	Ownable		
		Public	✓	Ownable
	setLiquidity	External	✓	onlyOwner
	deposit	External	✓	-
	mintRAIForStaking	External	✓	-
	receiveUSDT	External	✓	onlyAuthorized
	receiveLPFee	External	✓	-

	mintRAI	External	✓	onlyAuthorized
	removeLiquidityV3	External	✓	onlyOwner
	setAuthorizedContracts	External	✓	onlyOwner
	sqrt	Internal		
	valueOf	External		-
	setContract	External	✓	onlyOwner
	sweepToken	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
Context
_msgSender
_msgData
_contextSuffixLength
TreasuryContract
<Constructor>
setLiquidity
sweepToken
mintRAI
mintRAIForStaking
receiveLPFee
receiveUSDT
deposit
setAuthorizedContracts
onlyAuthorized
sqrt
valueOf
setContract
removeLiquidityV3
type
IERC20 (iface)
totalSupply
transferFrom
approve
allowance
balanceOf
transfer
_mint
IRaiToken (iface)
name
mint
approve
transfer
IPancakeSwapV3Router (iface)
ExactInputSingleParams
exactInputSingle
INonfungiblePositionManager (iface)
MintParams
mint
CollectParams
DecreaseLiquidityParams
decreaseLiquidity
collect
Ownable
<Constructor>
address
transferOwnership
Ownable.OwnableInvalidOwner
Ownable.OwnableUnauthorizedAccount
renounceOwnership
_transferOwnership
_msgSender
onlyOwner
_checkOwner
owner

Summary

RAIFI contract implements a treasury mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

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

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

About Cyberscope

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

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

The Cyberscope team

cyberscope.io