Cyberscope

Audit Report

Pyrand

December 2024

Repository https://github.com/Rana-usman-ul-haq/Pyrand-T/tree/main

Commit a4eb9d8b7ef82f8987dfc0625c9cec5c6cae55a6

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stop 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
●	ZD	Zero Division	Unresolved
●	CR	Code Repetition	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IDU	Inconsistent Decimal Usage	Unresolved
●	MFD	Misaligned Fee Distribution	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	NWES	Nonconformity with ERC-20 Standard	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RRA	Redundant Repeated Approvals	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSRS	Redundant SafeMath Require Statement	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

●	L09	Dead Code Elimination	Unresolved

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

Contract Name	PYRT
Repository	https://github.com/Rana-usman-ul-haq/Pyrand-T/tree/main
Commit	a4eb9d8b7ef82f8987dfc0625c9cec5c6cae55a6
Testing Deploy	https://testnet.bscscan.com/address/0x0147dad3b81ab1a910a1e5e8e9367f8e72e6d760
Symbol	PYRT
Decimals	18
Total Supply	100,000,000
Badge Eligibility	Must Fix Criticals

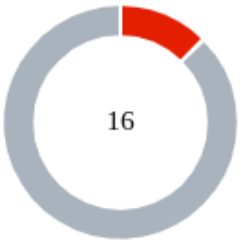
Audit Updates

Initial Audit	4 Dec 2024

Source Files

Filename	SHA256
contracts/PYRT.sol	4e5acdcbf3cee669c7e9466ef40d2acd251c0c1f074936016c8b83ef7d725fbf

Findings Breakdown



Critical	2
Medium	0
Minor / Informative	14

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	0	0	0	0
Minor / Informative	14	0	0	0

ST - Stop Transactions

Criticality	Critical
Location	contracts/PYRT.sol#L328
Status	Unresolved

Description

The contract owner has the authority to stop transactions, as described in detail in the `ZD` finding. As a result, the contract might operate as a honeypot.

Recommendation

The team is advised to follow the recommendations outlined in the `ZD` finding and implement the necessary steps to mitigate the identified risks, ensuring that the contract does not operate as a honeypot. Renouncing ownership will effectively eliminate the threats, but it is non-reversible.

ZD - Zero Division

Criticality	Critical
Location	contracts/PYRT.sol#L328
Status	Unresolved

Description

The contract uses 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 `_totalSellTax` variable can be set to zero.

```
uint256 marketingShare =
amount.mul(_sellMarketingFee).div(_totalSellTax);
uint256 communityShare =
amount.mul(_sellCommunityFee).div(_totalSellTax);
```

Recommendation

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.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L327
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.

```
   function sendETHToFee(uint256 amount) private {
        uint256 marketingShare =
amount.mul(_sellMarketingFee).div(_totalSellTax);
        uint256 communityShare =
amount.mul(_sellCommunityFee).div(_totalSellTax);
        uint256 otherShare = marketingShare.add(communityShare);
        uint256 devShare = amount.sub(otherShare);

        (bool callSuccess, ) = payable(_marketingWallet).call{value:
marketingShare}("");

        if (!callSuccess) {
        // Log the failure but do not revert the transaction
        emit TaxWalletPaymentRevert(_marketingWallet, marketingShare);
        }

        (bool callSuccessTwo, ) = payable(_communityWallet).call{value:
communityShare}("");

        if (!callSuccessTwo) {
        // Log the failure but do not revert the transaction
        emit TaxWalletPaymentRevert(_communityWallet, communityShare);
        }

        (bool callSuccessThree, ) = payable(_devWallet).call{value:
devShare}("");

        if (!callSuccessThree) {
        // Log the failure but do not revert the transaction
        emit TaxWalletPaymentRevert(_devWallet, devShare);
        }

    }
```

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	contracts/PYRT.sol#L197
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.

IDU - Inconsistent Decimal Usage

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L176
Status	Unresolved

Description

The contract is inconsistent in its usage of the decimals value by relying on `_decimals` in some instances and the `decimals()` function in others, despite both representing the same value. This inconsistency can cause confusion for developers and users, making the contract harder to maintain and understand.

```
uint256 public _taxSwapThreshold= 100_000 * 10**_decimals;
uint256 public maxWalletLimit = 5_000_000 * 10 ** decimals();
```

Recommendation

It is recommended to standardize the approach by using either `_decimals` or the `decimals()` function consistently throughout the contract. This will enhance readability and maintainability while ensuring clarity in the codebase.

MFD - Misaligned Fee Distribution

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L270,327
Status	Unresolved

Description

The contract is designed to calculate taxes for transfer, buy, and sell transactions based on their respective fee rates. However, the ETH obtained from swapping these fees is distributed solely based on the sell fee values, ignoring the buy and transfer fee values. This creates a discrepancy where, if the sell fee is lower than the buy or transfer fees, the contract will not account for the additional taxes collected from those transactions during swaps. As a result, the fee distribution mechanism becomes inconsistent and fails to accurately reflect the taxes collected, potentially leading to incorrect allocation of funds.

```
if(_FeeOnTransfers > 0) {
if(to != uniswapV2Pair && from != uniswapV2Pair) {
    taxAmount = amount.mul(_FeeOnTransfers).div(1000);
}
}

if(_totalBuyTax > 0) {
if (from == uniswapV2Pair && to != address(uniswapV2Router)) {
    taxAmount = amount.mul(_totalBuyTax).div(1000);
}
}

if(_totalSellTax > 0) {
if(to == uniswapV2Pair && from!= address(this) ){
    taxAmount = amount.mul(_totalSellTax).div(1000);
}
}

    function sendETHToFee(uint256 amount) private {
        uint256 marketingShare =
amount.mul(_sellMarketingFee).div(_totalSellTax);
        uint256 communityShare =
amount.mul(_sellCommunityFee).div(_totalSellTax);
        uint256 otherShare = marketingShare.add(communityShare);
        uint256 devShare = amount.sub(otherShare);
        ...
        }
```

Recommendation

It is recommended to modify the fee distribution logic to account for taxes collected from all transaction types—buy, sell, and transfer, when swapping and distributing the ETH. This ensures a fair and accurate allocation of funds based on the total taxes collected from all transaction categories.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L359
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 changeBuyFees(uint256 marketingFee, uint256 communityFee,
uint256 devFee) public onlyOwner {
        ...
    }

    function changeSellFees(uint256 marketingFee, uint256 communityFee,
uint256 devFee) public onlyOwner {
       ...
    }

    function whiteListFromFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = true;
    }

    function includeInFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = false;
    }

    function updateTaxWallets(address payable marketingCollection,
address payable communityCollection, address payable devCollection )
external onlyOwner {
        ...
    }

    function changeMaxWalletLimit(uint256 _limit) public onlyOwner{
        require(_limit > totalSupply().div(200),"Limit too low");
        maxWalletLimit = _limit;
    }

    function changeTransferFee(uint256 _transferTax) public onlyOwner {
        require(_transferTax <= 50, "Tax too high");
        _FeeOnTransfers = _transferTax;
    }

    function updateTaxSwapLimit(uint256 _taxLimit) public onlyOwner{
        require(_taxLimit > 0,"Limit too less");
        _taxSwapThreshold = _taxLimit;
    }
```

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.

NWES - Nonconformity with ERC-20 Standard

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L262
Status	Unresolved

Description

The contract is not fully conforming to the ERC20 Standard. Specifically, according to the standard, transfers of 0 values must be treated as normal transfers and fire the Transfer event. However the contract implements, a conditional check that prohibits transfers of 0 values.

This discrepancy between the contract's implementation and the ERC20 standard may lead to inconsistencies and incompatibilities with other contracts.

```
function _transfer(address from,address to,int256 amount) private {
...
require(amount > 0, "Transfer amount must be greater than zero");
...
}
```

Recommendation

The incorrect implementation of the ERC20 standard could potentially lead to problems when interacting with the contract, as other contracts or applications that expect the ERC20 interface may not behave as expected. The team is advised to review and revise the implementation of the transfer mechanism to ensure full compliance with the ERC20 standard. https://eips.ethereum.org/EIPS/eip-20.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L315
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.

```
path[0] = address(this);
path[1] = uniswapV2Router.WETH();
_approve(address(this), address(uniswapV2Router), tokenAmount);
uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
    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.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L289
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapTokensAtAmount` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

Additionally, the contract is designed to swap tokens for Ether under certain conditions. Specifically, the function `swapTokensForEth(contractTokenBalance)` is called. This means that the entire `contractTokenBalance` is swapped, rather than swapping only the amount specified by `_taxSwapThreshold`. As a result, the contract is swapping the full balance of tokens it holds at that moment, instead of just swapping up to the threshold limit defined by `_taxSwapThreshold`.

```
if (!inSwap && to == uniswapV2Pair && swapEnabled &&
contractTokenBalance>_taxSwapThreshold) {
    swapTokensForEth(contractTokenBalance);
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L317
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);
    // make the swap

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

Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/PYRT.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/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSRS - Redundant SafeMath Require Statement

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L24
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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/PYRT.sol#L182
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.

```
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	contracts/PYRT.sol#L137,153,155,156,157,158,160,161,162,163,165,167,168,169,172,173,174,175,176,392,397,402
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);
mapping (address => bool) public _isExcludedFromFee
uint256 public _buyMarketingFee = 32
uint256 public _buyCommunityFee = 9
uint256 public _buyDevFee = 9
uint256 public _totalBuyTax = 50
uint256 public _sellMarketingFee = 32
uint256 public _sellCommunityFee = 9
uint256 public _sellDevFee = 9
uint256 public _totalSellTax = 50
uint256 public _FeeOnTransfers = 0
address payable public _marketingWallet =
payable(0x5b7e9C0A4E350C2e861A7d2dA56F952066F276de)
address payable public _communityWallet =
payable(0xF3F76B63C72154dC01895192865Eaf5b785AA26e)
address payable public _devWallet =
payable(0x63349F6bAaeeA7272c05b62D0c933809b2539f98)

...
```

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	contracts/PYRT.sol#L308
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 min(uint256 a, uint256 b) private pure returns (uint256){
      return (a>b)?b:a;
    }
```

Recommendation

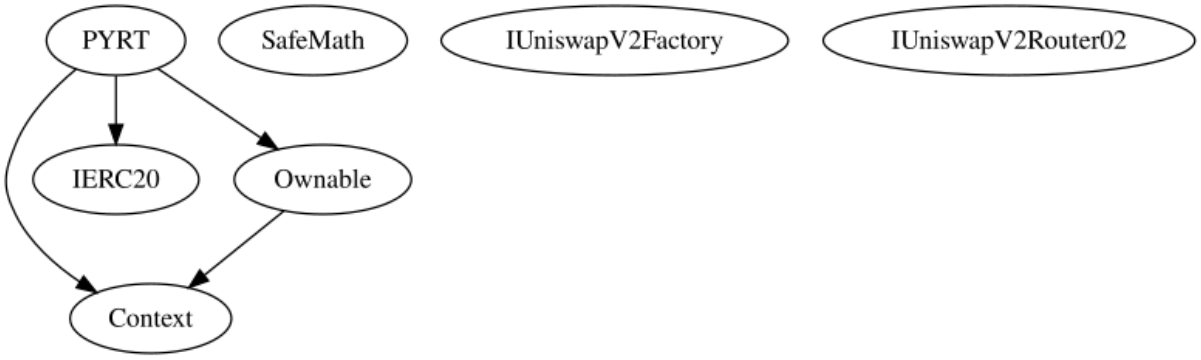
To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

Functions Analysis

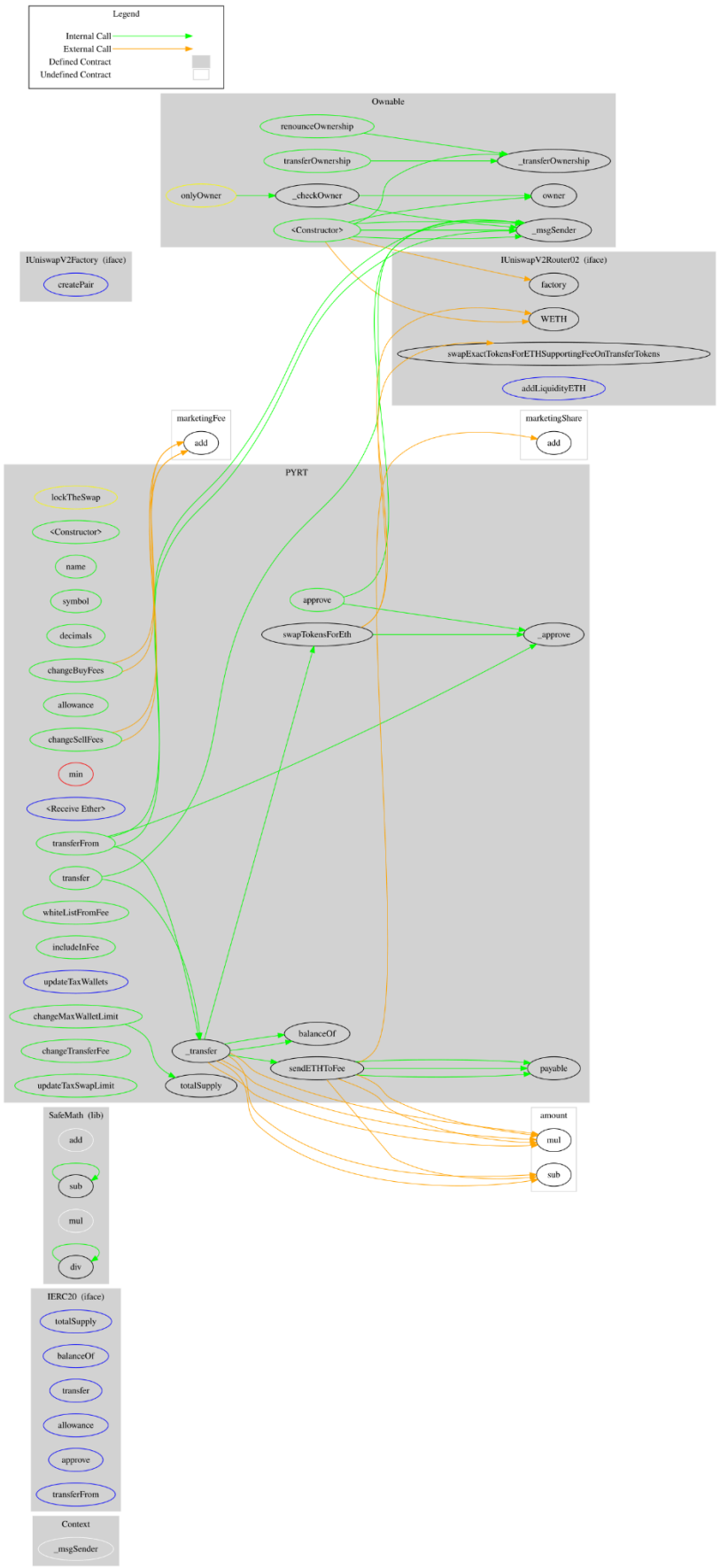
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
PYRT	Implementation	Context, IERC20, Ownable		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	_approve	Private	✓	
	_transfer	Private	✓	
	min	Private		
	swapTokensForEth	Private	✓	lockTheSwap
	sendETHToFee	Private	✓	
		External	Payable	-
	changeBuyFees	Public	✓	onlyOwner
	changeSellFees	Public	✓	onlyOwner

	whiteListFromFee	Public	✓	onlyOwner
	includeInFee	Public	✓	onlyOwner
	updateTaxWallets	External	✓	onlyOwner
	changeMaxWalletLimit	Public	✓	onlyOwner
	changeTransferFee	Public	✓	onlyOwner
	updateTaxSwapLimit	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
renounceOwnership
transferOwnership
_transferOwnership
onlyOwner
_checkOwner
owner
<Constructor>
_msgSender
IUniswapV2Factory (iface)
createPair
IUniswapV2Router02 (iface)
factory
WETH
swapExactTokensForETHSupportingFeeOnTransferTokens
addLiquidityETH
marketingFee
add
marketingShare
add
PYRT
lockTheSwap
<Constructor>
name
symbol
decimals
changeBuyFees
allowance
changeSellFees
min
<Receive Ether>
transferFrom
transfer
whiteListFromFee
includeInFee
updateTaxWallets
changeMaxWalletLimit
changeTransferFee
updateTaxSwapLimit
approve
swapTokensForEth
_approve
balanceOf
_transfer
sendETHToFee
payable
totalSupply
SafeMath (lib)
add
sub
mul
div
amount
mul
sub
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Context
_msgSender

Summary

Pyrand contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. There is also a limit of max 25% fees.

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