Audit Report

PYRAND

April 2024

Network	ETH
Address	0x67D8216cfC3CCDdA18CAf204034cEd50d4A7Ff8d
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://etherscan.io/address/0x67d8216cfc3ccdda18caf204034ced50d4a7ff8d

Audit Updates

Initial Audit	06 Apr 2024

Source Files

Filename	SHA256
PYRXPresale.sol	209853fbcbd1e06f0f1c88d748539ba91ba9d2ee757aae1c3c3f6a2202c70f81

Overview

The `PYRXPresale` smart contract is designed to facilitate a token presale event, managing the sale of tokens in exchange for Ether (ETH) and USD Tether (USDT). It utilizes the ERC20 standard for token transactions, employing interfaces like IERC20 for token interactions and SafeERC20 for secure token transfers. The contract is equipped with mechanisms to start and end the presale, handle token purchases, and allow token claims post-presale.

Token Sale Management

The contract supports two methods of purchasing tokens: using ETH and USDT. Purchasers can send ETH directly to the contract to buy tokens, or they can use USDT by calling the `buyTokensWithUSDT` function. The conversion of these currencies into tokens is guided by rates that are fetched from a Chainlink oracle for ETH and set manually for USDT.

Presale Control

The contract includes functions to start and end the presale, controlled exclusively by the contract owner. These state changes are essential to restrict or allow the buying and claiming of tokens according to the presale's schedule.

Token Pricing and Rate Updates

Prices and rates within the contract can be updated by the owner. This includes the ETH to USD conversion rate used for ETH purchases, the rate for USDT purchases, and the price of tokens in USD.

Token Deposits and Withdrawals

The owner can deposit tokens into the contract in preparation for the presale and withdraw unsold tokens after the presale. This functionality allows for the management of token inventory directly through the contract.

Claiming of Tokens

Participants can claim their purchased tokens after the presale has ended, ensuring that tokens are only distributed once the event is complete.

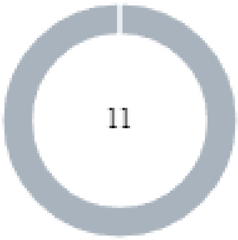
Financial Tracking

The contract tracks the amount of ETH and USDT raised during the presale, as well as the total number of tokens sold. It also maintains a record of each buyer's purchase through a mapping structure.

Utility Functions

Various utility functions are provided for both the owner and participants to check balances, view presale status, and retrieve current rates and prices.

Findings Breakdown



- Critical 0
- Medium 0
- Minor / Informative 11

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	11	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CCR	Contract Centralization Risk	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	PSCA	Presale Status Check Absence	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	USU	Unnecessary Struct Usage	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	PYRXPresale.sol#L724,760,765,815,850
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.

Specifically, the contract owner has to deposit the tokens and start the presale, so that users can purchase the tokens. Additionally, the owner has to end the presale, in order for users to be able to claim their tokens. Furthermore, the contract owner has the authority to heavily impact the functionality by changing the values of key contract parameters. Lastly, the contract interacts with an external contract, which is considered untrusted.

```
function startPresale() public onlyOwner{
    hasPresaleStarted = true;
}

function endPresale() public onlyOwner{
    hasPresaleEnded = true;
}

function changeRate(uint256 _usdtRate, uint256 _ethToUsd,
uint256 divider) public onlyOwner {
    require(_usdtRate > 0, "Rate cannot be 0");
    require(_ethToUsd > 0, "Rate cannot be 0");
    tokenPriceUSD = _ethToUsd;
    usdtRate = _usdtRate;
    ethDivider = divider;
}
```

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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	PYRXPresale.sol#L760,765,815,822,845,850
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 changeRate(uint256 _usdtRate, uint256 _ethToUsd,
uint256 divider) public onlyOwner {
    require(_usdtRate > 0, "Rate cannot be 0");
    require(_ethToUsd > 0, "Rate cannot be 0");
    tokenPriceUSD = _ethToUsd;
    usdtRate = _usdtRate;
    ethDivider = divider;
}
```

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.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	PYRXPresale.sol#L684
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
function getLatestPrice() public view returns (int) {
    (
        ,
        int price,
        ,
        ,
    ) = priceFeed.latestRoundData();
    // for BNB / USD price is scaled up by 10 ** 8
    return price / 1e8;
}
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

PSCA - Presale Status Check Absence

Criticality	Minor / Informative
Location	PYRXPresale.sol#L760,765
Status	Unresolved

Description

`hasPresaleStarted` and `hasPresaleEnded`, are two boolean flags, which respectively indicate whether the presale has started and whether it has concluded. They are managed by the contract owner through their respective functions. However, there is an oversight in the state management logic. The `endPresale` function does not check whether the presale has actually started before setting `hasPresaleEnded` to true.

```
function startPresale() public onlyOwner{
    hasPresaleStarted = true;
}

function endPresale() public onlyOwner{
    hasPresaleEnded = true;
}
```

Recommendation

It is recommended to implement additional checks within the `endPresale` function to ensure that the presale cannot be marked as ended unless it has officially started.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	PYRXPresale.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.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	PYRXPresale.sol#L760,765,845,850
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 changeWallet(address payable _wallet) external
onlyOwner {
    wallet = _wallet;
}

function changeRate(uint256 _usdtRate, uint256 _ethToUsd,
uint256 divider) public onlyOwner {
    require(_usdtRate > 0, "Rate cannot be 0");
    require(_ethToUsd > 0, "Rate cannot be 0");
    tokenPriceUSD = _ethToUsd;
    usdtRate = _usdtRate;
    ethDivider = divider;
}
```

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.

USU - Unnecessary Struct Usage

Criticality	Minor / Informative
Location	PYRXPresale.sol#L716
Status	Unresolved

Description

The `tokenBuyer` struct contains only the `tokensBought` field. The use of a struct for a single field is generally considered a bad practice and can lead to unnecessary complexity in the contract's data structure without providing any functional benefits. Structs are typically used to group together multiple related data items into a single entity, making the code more modular and readable. In this instance, however, the struct encapsulates only one data item, which could instead be directly associated with the buyer's address in a mapping, thereby simplifying the data handling and reducing the overhead associated with accessing struct elements.

```
struct tokenBuyer{
    uint256 tokensBought;
}
```

Recommendation

It is recommended to replace the struct `tokenBuyer` with a direct mapping of addresses to uint256, reflecting the number of tokens bought by each address. This approach enhances clarity and efficiency in the contract's design by eliminating an unnecessary layer of abstraction introduced by the struct.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	PYRXPresale.sol#L345,716,721,845,850
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 DOMAIN_SEPARATOR() external view returns (bytes32);

struct tokenBuyer{
        uint256 tokensBought;
    }
mapping (address => tokenBuyer) public Customer
address payable _wallet
uint256 _usdtRate
uint256 _ethToUsd
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	PYRXPresale.sol#L221,228,240,259,263,272,276,301,368,383,392,405
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 sendValue(address payable recipient, uint256 amount)
internal {
        require(address(this).balance >= amount, "Address:
insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value,
recipient may have reverted");
    }

function functionCall(address target, bytes memory data)
internal returns (bytes memory) {
        return functionCallWithValue(target, data, 0, "Address:
low-level call failed");
    }

...
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	PYRXPresale.sol#L846
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.

```
wallet = _wallet
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	PYRXPresale.sol#L321
Status	Unresolved

Description

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

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

```
assembly {
                let returndata_size := mload(returndata)
                revert(add(32, returndata), returndata_size)
            }
```

Recommendation

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

Functions Analysis

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

Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	verifyCallResultFromTarget	Internal		
	verifyCallResult	Internal		
	_revert	Private		
IERC20Permit	Interface			
	permit	External	✓	-
	nonces	External		-
	DOMAIN_SEPARATOR	External		-
SafeERC20	Library			
	safeTransfer	Internal	✓	

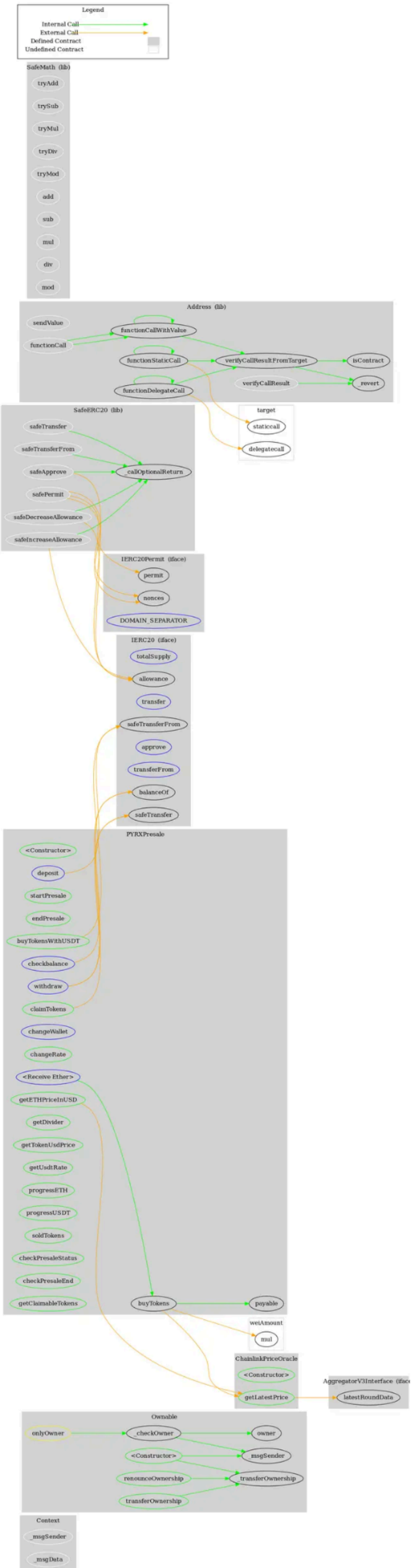
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	safeIncreaseAllowance	Internal	✓	
	safeDecreaseAllowance	Internal	✓	
	safePermit	Internal	✓	
	_callOptionalReturn	Private	✓	
SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		
ChainlinkPriceOracle	Implementation			

		Public	✓	-
	getLatestPrice	Public		-
AggregatorV3Interface	Interface			
	latestRoundData	External		-
PYRXPresale	Implementation	Ownable		
		Public	✓	-
		External	Payable	-
	startPresale	Public	✓	onlyOwner
	endPresale	Public	✓	onlyOwner
	buyTokens	Public	Payable	-
	buyTokensWithUSDT	Public	✓	-
	deposit	External	✓	onlyOwner
	withdraw	External	✓	onlyOwner
	claimTokens	Public	✓	-
	changeWallet	External	✓	onlyOwner
	changeRate	Public	✓	onlyOwner
	checkbalance	External		-
	getETHPriceInUSD	Public		-
	getDivider	Public		-
	getTokenUsdPrice	Public		-
	getUsdtRate	Public		-

	progressETH	Public		-
	progressUSDT	Public		-
	soldTokens	Public		-
	checkPresaleStatus	Public		-
	checkPresaleEnd	Public		-
	getClaimableTokens	Public		-

Inheritance Graph

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
SafeMath (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
add
sub
mul
div
mod
Address (lib)
sendValue
functionCall
functionCallWithValue
functionStaticCall
functionDelegateCall
verifyCallResultFromTarget
verifyCallResult
isContract
_revert
target
staticcall
delegatecall
SafeERC20 (lib)
safeTransfer
safeTransferFrom
safeApprove
safePermit
safeDecreaseAllowance
safeIncreaseAllowance
_callOptionalReturn
IERC20Permit (iface)
permit
nonces
DOMAIN_SEPARATOR
IERC20 (iface)
totalSupply
allowance
transfer
safeTransferFrom
approve
transferFrom
balanceOf
safeTransfer
PYRXPresale
<Constructor>
deposit
startPresale
endPresale
buyTokensWithUSDT
checkbalance
withdraw
claimTokens
changeWallet
changeRate
<Receive Ether>
getETHPriceInUSD
getDivider
getTokenUsdPrice
getUsdtRate
progressETH
progressUSDT
soldTokens
checkPresaleStatus
checkPresaleEnd
getClaimableTokens
buyTokens
payable
weiAmount
mul
ChainlinkPriceOracle
<Constructor>
getLatestPrice
latestRoundData
Ownable
onlyOwner
_checkOwner
owner
<Constructor>
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
Context
_msgSender
_msgData

Summary

PYRAND contract implements a presale 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

https://www.cyberscope.io