Audit Report

Kaspa Nexus

May 2024

Network	ETH
Address	0xe72D8576087F4606945d6327cD051Be6bca77956
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://etherscan.io/address/0xe72d8576087f4606945d6327cd051be6bca77956

Audit Updates

Initial Audit	30 Apr 2024 https://github.com/cyberscope-io/audits/blob/main/kspnx/v1/audit.pdf
Corrected Phase 2	08 May 2024

Source Files

Filename	SHA256
KSPNXPresale.sol	29f98646f43429ecd7ea52e58dabea63d1ec35440b64c8e499c2bbc3ba885f28

Overview

Buy Tokens Functionality

The KSPNXPresale contract allows users to purchase tokens using native currency (e.g., ETH), USDT, or USDC during a presale event. Users can specify a referral address to attribute part of their transaction for referral rewards. Additionally, purchasers have the option to stake their tokens directly upon buying. This functionality not only facilitates the acquisition of tokens but also integrates a staking feature to immediately begin earning potential rewards, enhancing user engagement and investment growth.

Unstake Tokens Functionality

The contract includes the `unStake` function that permits users to withdraw their staked tokens and any accrued rewards once the presale has concluded. This capability allows participants to claim their invested assets plus reward tokens based on the duration of their stake. The function is designed to cater to user needs, offering flexibility in managing investments and enhancing user control over their financial engagements within the platform.

Claim Tokens Functionality

Users can claim their purchased tokens through the `claimTokens` function, which is accessible after the presale has concluded. This function is crucial for allowing users to access their tokens, after the presale has ended, ensuring transparency and adherence to the presale terms.

Claim Referral Rewards Functionality

The contract also includes the `claimRefReward` function, enabling users to claim rewards earned through the referral system after the presale is over. This feature incentivizes users to promote the presale, thereby potentially increasing participation and investment.

Owner Functions

The contract grants the owner comprehensive control over critical parameters and functionalities, including the ability to update token address, and handle the presale stages and status. These owner-specific functions are essential for managing the presale dynamics and responding to changing market conditions or strategic needs. However, this level of control emphasizes the need for trust and transparency from the contract owner to prevent misuse or adverse unilateral decisions that could affect all stakeholders involved.

Roles

Owner

The owner can interact with the following functions:

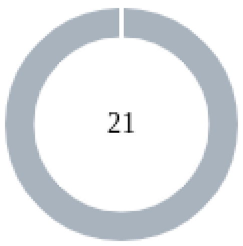
- function setPresaleStatus
- function endPresale
- function updateToken
- function whitelistAddresses
- function initiateTransfer
- function changeFundReciever
- function updatePriceFeed
- function setCurrentStage
- function transferTokens

Users

The users can interact with the following functions:

- function buyToken
- function buyTokenUSDT
- function buyTokenUSDC
- function unStake
- function claimTokens
- function claimRefReward

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	21

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	21	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ● Minor / Informative

Severity	Code	Description	Status
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	ITT	Inconsistent Token Tracking	Unresolved
●	IMU	Inefficient Memory Usage	Unresolved
●	ITC	Inefficient Token Calculation	Unresolved
●	MC	Missing Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MU	Modifiers Usage	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	RED	Redundant Event Declaration	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved

●	OCTD	Transfers Contract's Tokens	Unresolved
●	UVD	Unnecessary Variable Declaration	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

CR - Code Repetition

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L261,307,355
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.

Specifically the `buyToken` , `buyTokenUSDT` and `buyTokenUSDC` functions share similar code segments.

```
   function buyToken(bool _isStake, address _refAddress) public
payable {
        ...
    }

    function buyTokenUSDT(
        uint256 amount,
        bool _isStake,
        address _refAddress
    ) public {
        ...
    }

    function buyTokenUSDC(
        uint256 amount,
        bool _isStake,
        address _refAddress
    ) public {
        ...
    }
```

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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L482,486,517,556
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.

Additionally, the contract is currently implementing a `require` statement to limit the number of tokens purchased in each phase based on the total tokens available for that stage. However, when the limit of tokens available for a stage is reached, the contract does not automatically proceed to the next phase. Instead, it relies on the contract owner to manually increment the current stage to allow the presale to continue. This reliance could introduce delays or potential management errors in the presale process.

```
   function setPresaleStatus(bool _status) external onlyOwner {
        presaleStatus = _status;
    }

    function endPresale() external onlyOwner {
        require(!isPresaleEnded, "Already ended");
        isPresaleEnded = true;
    }
    function whitelistAddresses(
        address[] memory _addresses,
        uint256[] memory _tokenAmount,
        uint256[] memory _refAmount
    ) external onlyOwner {
    ...
    }
    function setCurrentStage(uint256 _stageNum) public
onlyOwner {
        currentStage = _stageNum;
    }
```

```
require(
    phases[currentStage].tokenSold + numberOfTokens <=
        phases[currentStage].tokensToSell,
    "Phase Limit Reached"
);
```

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.

Additionally, it is recommended to consider using a functionality that automatically increments the stage to the next one once the available tokens of the current phase are reached. Automating this process can enhance the efficiency of the presale stages, reduce the dependency on manual intervention, and ensure a smoother transaction flow for participants. This change would also mitigate potential risks associated with delayed or missed manual updates, contributing to a more reliable and user-friendly presale experience.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L498,508,513
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

Additionally, the contract is currently equipped with the `updateToken` function that allows the contract owner to change the main token address at any time. This level of control poses significant risks as it affects the values and the balances within the contract and impacts many critical calculations related to token operations. Since the token address can be altered unilaterally by the owner, it could lead to disruptions in the contract’s operations and potentially compromise the security and integrity of the presale process.

```
uint256 numberOfTokens = (ethToUsd * getPhaseDetail(phaseId)) /
(1e8);
...
uint256 numberOfTokens = (_amount * getPhaseDetail(phaseId)) /
(1e6);
```

```
function updateToken(address _token) external onlyOwner {
    mainToken = IERC20(_token);
}
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

Additionally it is recommended to consider removing the `updateToken` function from the contract to prevent the potential risks associated with unilateral changes to the main token address by the contract owner. Should there be a necessary reason to retain this capability, it is crucial to implement strict checks within the function to ensure the new token address shares identical characteristics with the current token, such as decimals and total supply. This approach will help maintain the integrity and stability of the contract's operations.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L221
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.

```
distrubuteAbleReward
```

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.

ITT - Inconsistent Token Tracking

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L285,335,383
Status	Unresolved

Description

The contract is tracking the tokens sold through the `soldToken` variable to monitor the number of tokens distributed during sales. However, it lacks a mechanism to track the total tokens assigned as referral rewards, which are computed and allocated whenever a sale involves a referral. This discrepancy results in inconsistent data handling, as the contract only tracks tokens in specific scenarios (direct sales) and not universally (including referral rewards).

```
    if (referral[msg.sender] == address(0) &&
isExist[_refAddress]) {
        referral[msg.sender] = _refAddress;
        referralCount[_refAddress] += 1;
        users[_refAddress].refReward +=
            (numberOfTokens * RefRewardPercent) /
            percentDivider;
}
soldToken = soldToken + (numberOfTokens);
```

Recommendation

It is recommended to introduce a variable to consistently track the total tokens assigned as referral rewards alongside the `soldToken` variable. This addition will provide a comprehensive overview of all tokens distributed through the contract, encompassing both direct sales and referral rewards. Implementing this change will ensure data consistency and enhance the transparency of token distribution, which is crucial for accurate auditing and stakeholder confidence.

IMU - Inefficient Memory Usage

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L474
Status	Unresolved

Description

The contract is currently designed such that the `getPhaseDetail` function retrieves phase data as a memory object. This approach is inefficient because the function merely returns a single variable, `tokenPerUsdPrice`, without modifying the phase object. The process of copying the entire phase into memory for accessing a single attribute leads to unnecessary gas consumption which could be avoided.

```
function getPhaseDetail(uint256 phaseInd)
    public
    view
    returns (uint256 priceUsd)
{
    Phase memory phase = phases[phaseInd];
    return (phase.tokenPerUsdPrice);
}
```

Recommendation

It is recommended that the `getPhaseDetail` function should be optimized by directly accessing the `tokenPerUsdPrice` from the state storage instead of copying the entire phase into memory. By directly accessing the variable from the state, this change would reduce gas costs and enhance the efficiency of the function. This approach not only ensures consistency in accessing phase details across the contract but also minimizes memory usage, thus optimizing overall contract performance.

ITC - Inefficient Token Calculation

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L272,285,322,335,370,383
Status	Unresolved

Description

The contract is currently calculating the total tokenSold multiple times within different parts of its logic, specifically during checks and actual updates. It increments the `tokenSold` variable by the `numberOfTokens` amount in separate operations, which not only makes the code less efficient but also increases gas consumption. Each addition operation on state variables costs gas, and by performing this increment operation multiple times, the contract unnecessarily inflates the transaction cost.

```
require(
    phases[currentStage].tokenSold + numberOfTokens <=
        phases[currentStage].tokensToSell,
    "Phase Limit Reached"
);
...
soldToken = soldToken + (numberOfTokens);
phases[currentStage].tokenSold += numberOfTokens;
```

Recommendation

It is recommended to optimize the token sales calculation by consolidating the increment operations into a single pre-calculated step prior to the `require` statement. This approach will reduce the redundant calculations and updates to the `tokenSold` variable, thereby decreasing the gas consumption associated with multiple state changes. By calculating the total tokens sold once and using this pre-computed value for both validation and update purposes, the contract can achieve more efficient and cost-effective execution, enhancing overall performance.

MC - Missing Check

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L556
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 `setCurrentStage` function is missing a check to verify that the `_stageNum` exist.

```
    function setCurrentStage(uint256 _stageNum) public onlyOwner
{
        currentStage = _stageNum;
    }
```

Recommendation

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

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L482,486
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 setPresaleStatus(bool _status) external onlyOwner {
    presaleStatus = _status;
}
function endPresale() external onlyOwner {
    require(!isPresaleEnded, "Already ended");
    isPresaleEnded = true;
}
```

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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L262,312,360,415,435,447
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(_refAddress != msg.sender, "cant ref yourself");
require(!isPresaleEnded, "Presale ended!");
require(presaleStatus, " Presale is Paused, check back later");
require(isPresaleEnded, "Presale has not ended yet");
require(isExist[msg.sender], "User don't exist");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L497
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.

```
uint256 ethToUsd = (_amount * (getLatestPrice())) / (1 ether);
```

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.

RED - Redundant Event Declaration

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L212
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 UpdatePrice(uint256 _oldPrice, uint256 _newPrice);
```

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.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L483
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 setPresaleStatus(bool _status) external onlyOwner {
        presaleStatus = _status;
    }
```

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.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L215,426,442,454
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
mainToken = _token;
...
mainToken.transfer(msg.sender, _reward);
...
mainToken.transfer(msg.sender, userStake.stakedTokens);
...
mainToken.transfer(msg.sender, claimAmount);
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L562
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 `transferTokens` function.

```
    function transferTokens(IERC20 token, uint256 _value)
external onlyOwner {
        token.transfer(fundReceiver, _value);
    }
```

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.

UVD - Unnecessary Variable Declaration

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L498
Status	Unresolved

Description

The contract is currently designed to declare an intermediate variable `numberOfTokens` within the function to hold the result of the calculation before returning this value. Specifically, the calculation `(ethToUsd * phases[phaseId].tokenPerUsdPrice) / (1e8)` is assigned to `numberOfTokens`, which is then immediately returned. This additional step of storing the calculation result in a variable before the return is redundant and slightly increases the gas cost of each transaction due to the extra operation involved.

```
uint256 numberOfTokens = (ethToUsd *
phases[phaseId].tokenPerUsdPrice) /
      (1e8);
return numberOfTokens;
```

Recommendation

It is recommended to directly return the result of the calculation instead of first assigning it to an intermediate variable. By modifying the function to return the calue directly, the contract can reduce gas consumption and streamline the code execution. This change enhances efficiency and clarity in the contract's logic without impacting the functionality.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L134,135
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.

```
IERC20 public USDT =
IERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7)
IERC20 public USDC =
IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48)
```

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	KSPNXPresale.sol#L134,135,146,147,148,169,261,309,310,357,358,403,412,457,482,492,503,513,518,519,520,534,543,548,556,561
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.

```
IERC20 public USDT =
IERC20(0xdAC17F958D2ee523a2206206994597C13D831ec7)
IERC20 public USDC =
IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48)
uint256 constant year = 365 days
uint256 constant percentDivider = 100_00
uint256 constant RefRewardPercent = 10_00
address[] public UsersAddresses
bool _isStake
address _refAddress
uint256 _amount
uint256 _index
address _user
bool _status
address _token
address[] memory _addresses

...
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	KSPNXPresale.sol#L216,544
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.

```
fundReceiver = payable(_fundReceiver)
fundReceiver = payable(_addr)
```

Recommendation

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

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	KSPNXPresale.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 ^0.8.17;
```

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	KSPNXPresale.sol#L319,367,421,425,441,453,562
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.

```
USDT.transferFrom(msg.sender, address(this), amount)
USDC.transferFrom(msg.sender, address(this), amount)
mainToken.transfer(msg.sender, _reward)
mainToken.transfer(msg.sender, userStake.stakedTokens)
mainToken.transfer(msg.sender, claimAmount)
token.transfer(fundReceiver, _value)
```

Recommendation

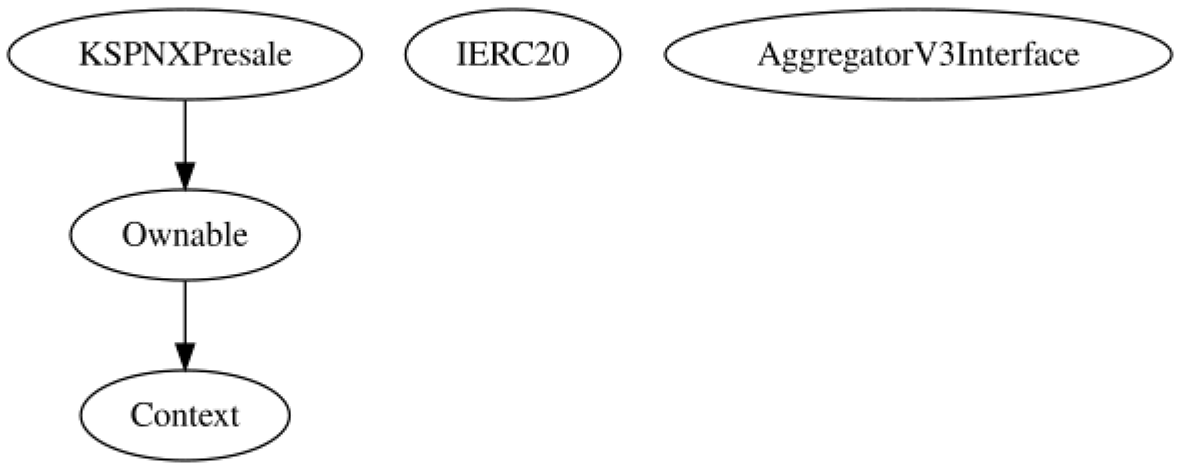
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
KSPNXPresale	Implementation	Ownable		
		Public	✓	-
	getLatestPrice	Public		-
	buyToken	Public	Payable	-
	buyTokenUSDT	Public	✓	-
	buyTokenUSDC	Public	✓	-
	stake	Internal	✓	
	unStake	Public	✓	-
	claimTokens	External	✓	-
	claimRefReward	Public	✓	-
	calculateReward	Public		-
	getPhaseDetail	Public		-
	setPresaleStatus	External	✓	onlyOwner
	endPresale	External	✓	onlyOwner
	nativeToToken	Public		-
	usdtToToken	Public		-
	updateToken	External	✓	onlyOwner
	whitelistAddresses	External	✓	onlyOwner
	initiateTransfer	External	✓	onlyOwner

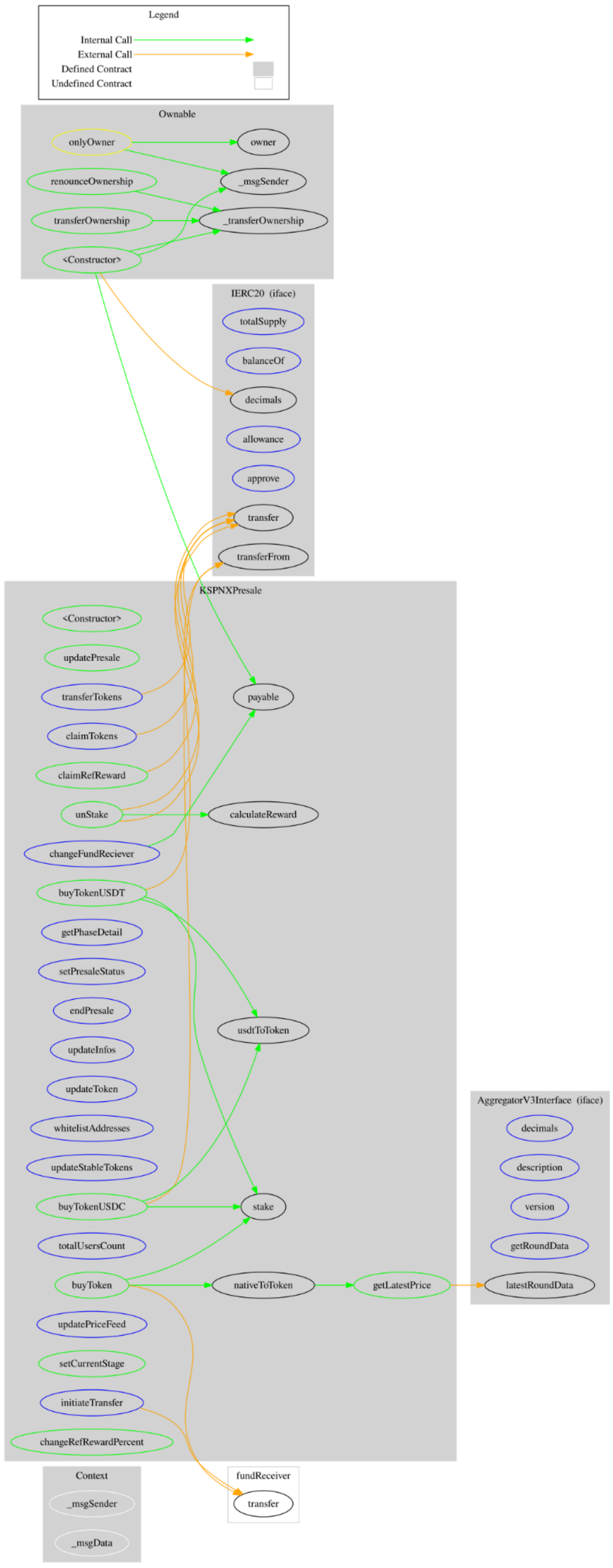
	totalUsersCount	External		-
	changeFundReciever	External	✓	onlyOwner
	updatePriceFeed	External	✓	onlyOwner
	setCurrentStage	Public	✓	onlyOwner
	transferTokens	External	✓	onlyOwner

Inheritance Graph



$$\begin{tikzcd}
\text{KSPNXPresale} \arrow[d] & \text{IERC20} & \text{AggregatorV3Interface} \\
\text{Ownable} \arrow[d] & & \\
\text{Context} & &
\end{tikzcd}$$

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
onlyOwner
owner
renounceOwnership
_msgSender
transferOwnership
_transferOwnership
<Constructor>
IERC20 (iface)
totalSupply
balanceOf
decimals
allowance
approve
transfer
transferFrom
KSPNXPresale
<Constructor>
updatePresale
transferTokens
payable
claimTokens
claimRefReward
unStake
calculateReward
changeFundReciever
buyTokenUSDT
getPhaseDetail
setPresaleStatus
endPresale
usdtToToken
updateInfos
updateToken
whitelistAddresses
updateStableTokens
buyTokenUSDC
stake
totalUsersCount
buyToken
nativeToToken
getLatestPrice
updatePriceFeed
setCurrentStage
initiateTransfer
changeRefRewardPercent
AggregatorV3Interface (iface)
decimals
description
version
getRoundData
latestRoundData
Context
_msgSender
_msgData
fundReceiver
transfer

Summary

The Kaspa Nexus contract implements a presale mechanism for distributing tokens using cryptocurrencies like the native token, USDT, and USDC, with options for staking and rewards based on the duration of the stake. This audit investigates security issues, business logic concerns, and potential improvements to ensure the contract operates efficiently, securely, and in alignment with best practices for smart contract development.

Disclaimer

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