Cyberscope

Audit Report

TG.Bet Presale

January 2024

SHA256 d679da84e555471da684418bbf58f23b4614b05bd143ebfedb31755238025c48

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0x8c25089a2dfad4337586eb347ffe71e3cb132759

Audit Updates

Initial Audit	18 Jan 2024

Source Files

Filename	SHA256
contracts/PresaleV1.sol	d679da84e555471da684418bbf58f23b4614b05bd143ebfedb31755238025c48
@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol	2d3d7dc6e116cb8ebb8517208141cb3d0950b337a285f15f8476ec3df29d824e
@openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol	db92fc1b515decad3a783b1422190877d2d70b907c6e36fb0998d9465aee42db
@openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol	78a6bc84bbb417f0d8a6b12e181e0f783151774f4f0c054c5d3f920e70d69f8c
@openzeppelin/contracts-upgradeable/security/ReentrancyGuardUpgradeable.sol	82789e6bdc1d6b5d8348a741281ec7a1e46c602017e1e4b7ae805bd55bfed26b
@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol	a2c4e5c274a586f145d278293ae33198cd8f412ab7e6d26f2394c8949b32b24b
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol	2d9e57d2a4b0775334be2968019c1939377d45b69e8b724fe6bb80af47e28419

Overview

The PresaleV1 smart contract facilitates the purchase of tokens using USDC, USDT, and ETH, with the price of ETH being sourced from an oracle. The contract includes features such as setting sale times, buying tokens with different payment methods, staking tokens, and withdrawing remaining tokens. Additionally, it allows the owner to disperse tokens to multiple addresses.

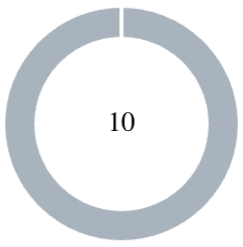
Functionality: The contract provides the expected functionalities, allowing users to buy tokens with various payment methods (USDT, USDC, ETH), stake tokens, and withdraw remaining tokens.

Oracle Interaction: The contract uses an external oracle (Aggregator) to fetch the latest ETH/USDT price.

Centralization: The contract includes functions to set sale times, update max tokens to buy, set the payment wallet, and set the token price. These functions provide flexibility to adapt to changing conditions.

Owner Dispersal Mechanism: The dispersal function allows the owner to distribute tokens among multiple addresses. Ensure that this function is used responsibly and in accordance with the project's intentions.

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	10

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	10	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	SETCCA	Start End Time Check Conditional Absence	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L345
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 _buyWithUSDT(uint256 amount, bytes32 invitation) internal {}
function _buyWithUSDC(uint256 amount, bytes32 invitation) internal {}
```

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	contracts/PresaleV1.sol#L479
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.

- withdrawRemainingTokens()
- setMaxTokensToBuy()
- setPaymentWallet()
- setTokenPrice()

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	contracts/PresaleV1.sol#L130
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.

```
uint256 total = saleToken.balanceOf(_msgSender());
uint256 amount = total / 100;
saleToken.transferFrom(_msgSender(), address(this), amount * 40);
saleToken.transferFrom(_msgSender(), address(stakingManagerInterface), amount
* 20);
saleToken.transferFrom(_msgSender(), _accountAddresses[0], amount * 10);
saleToken.transferFrom(_msgSender(), _accountAddresses[1], amount * 10);
saleToken.transferFrom(_msgSender(), _accountAddresses[2], amount * 5);
saleToken.transferFrom(_msgSender(), _accountAddresses[3], amount * 5);
saleToken.transferFrom(_msgSender(), _accountAddresses[4], amount * 5);
saleToken.transferFrom(_msgSender(), _accountAddresses[5], amount * 5);
```

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.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L347
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.

```
uint256 price = usdPrice / (10 ** 12);
...
uint256 price = usdPrice / (10 ** 12);
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The number decimals should be fetched from the corresponding address rather than setting a fixed value. 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.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L145
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 (uint256) {
    (, int256 price, , , ) = aggregatorInterface.latestRoundData();
    price = (price * (10 ** 10));
    return uint256(price);
}
```

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.

For instance, the chainlink API provides a method that returns the decimal precision https://docs.chain.link/data-feeds/api-reference#decimals

SETCCA - Start End Time Check Conditional Absence

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L490
Status	Unresolved

Description

In the setSaleTimes function, there is a conditional check for the start and end times; however, if the owner sets only the start time without specifying the end time, the invariant check for the end time may be violated. The current implementation does not ensure that the end time is later than the start time in such a scenario.

```
if (_startTime > 0) {
    require(block.timestamp < startTime, "Sale already started");
    require(block.timestamp < _startTime, "Sale time in past");
    uint256 prevValue = startTime;
    startTime = _startTime;
    emit SaleTimeUpdated(
        bytes32("START"),
        prevValue,
        _startTime,
        block.timestamp
    );
}
if (_endTime > 0) {
    require(block.timestamp < endTime, "Sale already ended");
    require(_endTime > startTime, "Invalid endTime");
    uint256 prevValue = endTime;
    endTime = _endTime;
    emit SaleTimeUpdated(
        bytes32("END"),
        prevValue,
        _endTime,
        block.timestamp
    );
}
```

Recommendation

To address this issue, consider modifying the code to include an additional check when only the start time is set. This check should ensure that if the end time is already set, it should be later than the new start time. This modification ensures that the invariant of the end time being later than the start time is maintained even when only the start time is updated.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L42,43,88,89,90,91,92,93,94,95,96,97,127,491,492,521,532,541
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.

```
IERC20Upgradeable public USDTInterface
IERC20Upgradeable public USDCInterface
address _oracle
address _usdt
address _usdc
address _saleToken
address _stakingContract
uint256 _tokenPrice
uint256 _startTime
uint256 _endTime
uint256 _maxTokensToBuy
address _paymentWallet
address[] calldata _accountAddresses
address _newPaymentWallet
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L542
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
tokenPrice = _tokenPrice
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L121
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.

```
paymentWallet = _paymentWallet
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/PresaleV1.sol#L132,133,134,135,136,137,138,139
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.

```
saleToken.transferFrom(_msgSender(), address(this), amount * 40)
saleToken.transferFrom(_msgSender(), address(stakingManagerInterface), amount
* 20)
saleToken.transferFrom(_msgSender(), _accountAddresses[0], amount * 10)
saleToken.transferFrom(_msgSender(), _accountAddresses[1], amount * 10)
saleToken.transferFrom(_msgSender(), _accountAddresses[2], amount * 5)
saleToken.transferFrom(_msgSender(), _accountAddresses[3], amount * 5)
saleToken.transferFrom(_msgSender(), _accountAddresses[4], amount * 5)
saleToken.transferFrom(_msgSender(), _accountAddresses[5], amount * 5)
```

Recommendation

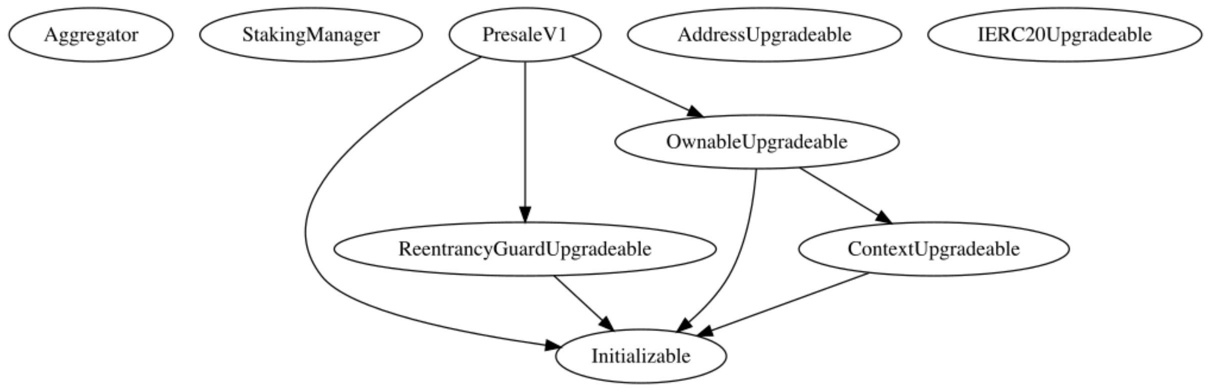
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

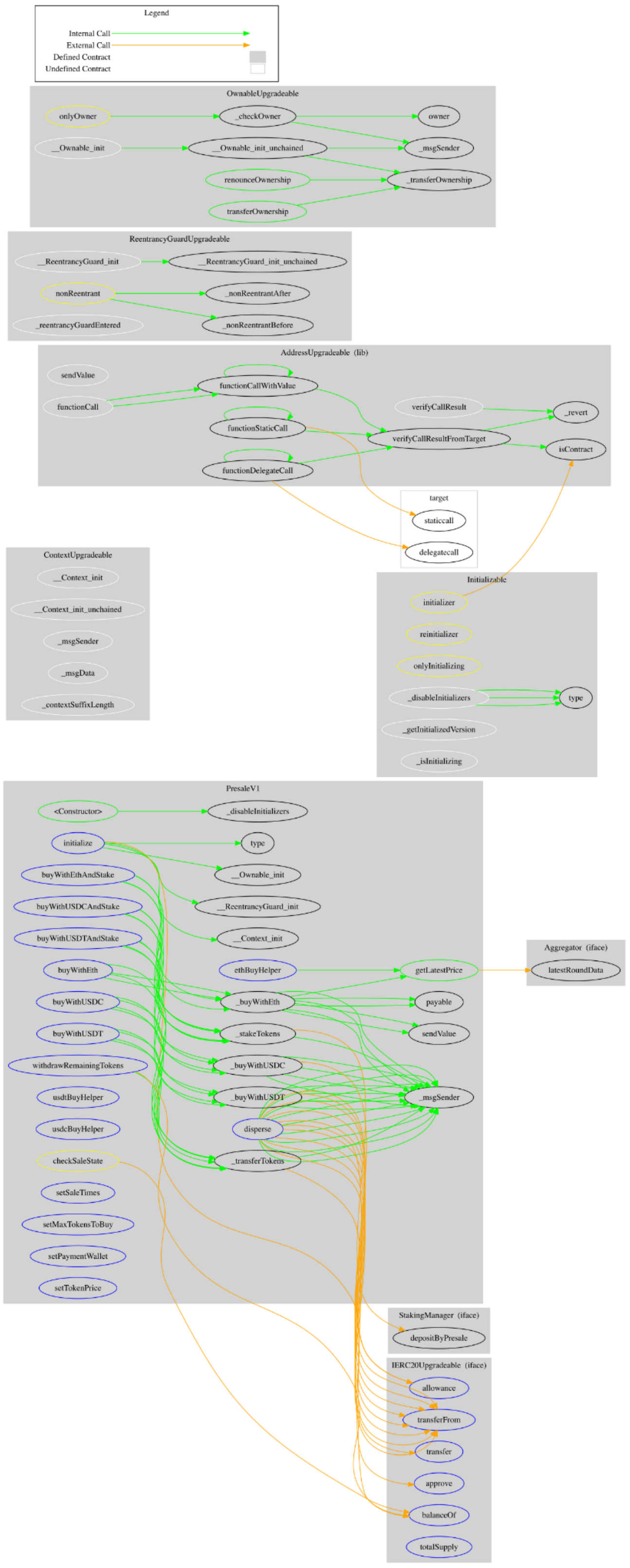
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Aggregator	Interface			
	latestRoundData	External		-
StakingManager	Interface			
	depositByPresale	External	✓	-
PresaleV1	Implementation	Initializable, ReentrancyGuardUpgradeable, OwnableUpgradeable		
		Public	✓	-
	initialize	External	✓	initializer
	disperse	External	✓	onlyOwner
	getLatestPrice	Public		-
	buyWithUSDT	External	✓	checkSaleState
	buyWithUSDT	External	✓	checkSaleState
	buyWithUSDC	External	✓	checkSaleState
	buyWithUSDC	External	✓	checkSaleState
	buyWithEth	External	Payable	checkSaleState nonReentrant
	buyWithEth	External	Payable	checkSaleState nonReentrant

	buyWithUSDTAndStake	External	✓	checkSaleState
	buyWithUSDTAndStake	External	✓	checkSaleState
	buyWithUSDCAndStake	External	✓	checkSaleState
	buyWithUSDCAndStake	External	✓	checkSaleState
	buyWithEthAndStake	External	Payable	checkSaleState nonReentrant
	buyWithEthAndStake	External	Payable	checkSaleState nonReentrant
	_buyWithUSDT	Internal	✓	
	_buyWithUSDC	Internal	✓	
	_buyWithEth	Internal	✓	
	_transferTokens	Internal	✓	
	_stakeTokens	Internal	✓	
	ethBuyHelper	External		-
	usdtBuyHelper	External		-
	usdcBuyHelper	External		-
	sendValue	Internal	✓	
	withdrawRemainingTokens	External	✓	onlyOwner
	setSaleTimes	External	✓	onlyOwner
	setMaxTokensToBuy	External	✓	onlyOwner
	setPaymentWallet	External	✓	onlyOwner
	setTokenPrice	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
OwnableUpgradeable
onlyOwner
_checkOwner
owner
__Ownable_init
__Ownable_init_unchained
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
ReentrancyGuardUpgradeable
__ReentrancyGuard_init
__ReentrancyGuard_init_unchained
nonReentrant
_nonReentrantAfter
_reentrancyGuardEntered
_nonReentrantBefore
AddressUpgradeable (lib)
sendValue
functionCallWithValue
functionCall
verifyCallResult
_revert
functionStaticCall
verifyCallResultFromTarget
isContract
functionDelegateCall
target
staticcall
delegatecall
ContextUpgradeable
__Context_init
__Context_init_unchained
_msgSender
_msgData
_contextSuffixLength
Initializable
initializer
reinitializer
onlyInitializing
_disableInitializers
type
_getInitializedVersion
_isInitializing
PresaleV1
<Constructor>
_disableInitializers
initialize
type
buyWithEthAndStake
__Ownable_init
buyWithUSDCAndStake
__ReentrancyGuard_init
buyWithUSDTAndStake
__Context_init
buyWithEth
ethBuyHelper
getLatestPrice
buyWithUSDC
_buyWithEth
payable
buyWithUSDT
_stakeTokens
sendValue
withdrawRemainingTokens
_buyWithUSDC
usdtBuyHelper
_buyWithUSDT
_msgSender
usdcBuyHelper
disperse
checkSaleState
_transferTokens
setSaleTimes
setMaxTokensToBuy
setPaymentWallet
setTokenPrice
Aggregator (iface)
latestRoundData
StakingManager (iface)
depositByPresale
IERC20Upgradeable (iface)
allowance
transferFrom
transfer
approve
balanceOf
totalSupply

Summary

The PresaleV1 contract facilitates token presale with USDC, USDT, and ETH. It integrates key functionalities for managing sale times, buying, staking, and withdrawals. This audit investigates security issues, business logic concerns and potential improvements.

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