Cyberscope

Audit Report

PLOUTOS

July 2024

Repository https://github.com/Ploutoslabs/Ploutoslab-contract

Commit 6f96a8f98c987e08c17c890ba45efc37adacbb78

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Unresolved
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	PTPM	Potential Token Price Manipulation	Unresolved
●	TCI	Token Cap Issue	Unresolved
●	TSD	Total Supply Diversion	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IAC	Immediate Allocation Claim	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MC	Misleading Comment	Unresolved
●	MU	Modifiers Usage	Unresolved
●	NMU	NonReentrant Modifier usage	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	PSI	Presale Status Initialization	Unresolved
●	RC	Redundant Check	Unresolved
●	TNE	Typographical Naming Error	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Analysis **1**
Diagnostics **2**
Table of Contents **4**
Review **6**
Audit Updates 6
Source Files 6
Overview **8**
Findings Breakdown **10**
MT - Mints Tokens 11
Description 11
Recommendation 12
PTPM - Potential Token Price Manipulation 13
Description 13
Recommendation 14
TCI - Token Cap Issue 15
Description 15
Recommendation 17
TSD - Total Supply Diversion 18
Description 18
Recommendation 18
CCR - Contract Centralization Risk 19
Description 19
Recommendation 20
IAC - Immediate Allocation Claim 21
Description 21
Recommendation 23
IDI - Immutable Declaration Improvement 24
Description 24
Recommendation 24
MC - Misleading Comment 25
Description 25
Recommendation 25
MU - Modifiers Usage 26
Description 26
Recommendation 26
NMU - NonReentrant Modifier usage 27
Description 27
Recommendation 29
PTRP - Potential Transfer Revert Propagation 30

Description 30
Recommendation 31
PSI - Presale Status Initialization 32
Description 32
Recommendation 32
RC - Redundant Check 33
Description 33
Recommendation 35
TNE - Typographical Naming Error 36
Description 36
Recommendation 36
L04 - Conformance to Solidity Naming Conventions 37
Description 37
Recommendation 37
L16 - Validate Variable Setters 38
Description 38
Recommendation 38
L19 - Stable Compiler Version 39
Description 39
Recommendation 39
Functions Analysis **40**
Inheritance Graph **41**
Flow Graph **42**
Summary **43**
Disclaimer **44**
About Cyberscope **45**

Review

Contract Name	Ploutos
Repository	https://github.com/Ploutoslabs/Ploutoslab-contract
Commit	6f96a8f98c987e08c17c890ba45efc37adacbb78
Testing Deploy	https://testnet.bscscan.com/address/0x4907f4509420df0f593acb3a61d1baed012f5a86
Symbol	PLTL
Decimals	9
Total Supply	210,000,461.53
Badge Eligibility	Must Fix Criticals

Audit Updates

Initial Audit	03 Jul 2024

Source Files

Filename	SHA256
contracts/Ploutos.sol	2c213eec3869e1870ed52fddf156fb533b0757b4591cbf9001d845e096f5686c
@openzeppelin/contracts/utils/ReentrancyGuard.sol	8d0bac508a25133c9ff80206f65164cef959ec084645d1e7b06050c2971ae0fc
@openzeppelin/contracts/utils/Context.sol	847fda5460fee70f56f4200f59b82ae622bb03c79c77e67af010e31b7e2cc5b6

@openzeppelin/contracts/token/ERC20/IERC20.sol	6f2faae462e286e24e091d7718575179644dc60e79936ef0c92e2d1ab3ca3cee
@openzeppelin/contracts/token/ERC20/ERC20.sol	ddff96777a834b51a08fec26c69bb6ca2d01d150a3142b3fdd8942e07921636a
@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol	1d079c20a192a135308e99fa5515c27acfbb071e6cdb0913b13634e630865939
@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol	cb15f210495f2119cfec53e32738ddb23469d414c45a2d7444c2181a9940bbed
@openzeppelin/contracts/interfaces/draft-IERC6093.sol	4aea87243e6de38804bf8737bf86f750443d3b5e63dd0fd0b7ad92f77cdbd3e3
@openzeppelin/contracts/access/Ownable.sol	38578bd71c0a909840e67202db527cc6b4e6b437e0f39f0c909da32c1e30cb81

Overview

The Ploutos smart contract is an ERC20 token implementation utilizing the OpenZeppelin library, specifically incorporating the ERC20Capped extension to enforce a maximum token supply. The contract is designed to manage token allocations through a structured process involving airdrops, presale purchases, and subsequent claims by users. It includes several key functionalities:

Airdrop Functionality

Users can participate in an airdrop by paying a specified fee. Upon participation, they receive an immediate allocation of tokens and a larger total amount set for future claims.

Presale Functionality

Users can purchase tokens during a presale period, which is controlled by the contract owner or an admin. The purchase rate is determined by a presale rate, and users receive a portion of their purchased tokens immediately, with the remainder available for future claims over time.

Allocation and Claim Mechanism

The contract manages token allocations through a structured process. Allocations are recorded in a mapping and can be claimed by users at specified intervals. The claiming process involves calculating the number of periods that have elapsed and determining the amount of tokens available for claim based on this calculation.

Administrative Controls

The contract includes various administrative controls, allowing the owner or an admin to set the presale rate, start or stop the presale, and allocate tokens to users. These controls are protected by access restrictions to ensure only authorized accounts can perform these actions.

Events and Emissions

The contract emits several events to provide transparency and traceability of actions such as airdrop claims, presale purchases, and allocation increases. These events facilitate monitoring and auditing of the contract's activities.

Token Characteristics

The Ploutos token (PLTL) is designed with a fixed maximum supply and a specified number of decimals. The contract also includes functions to return the total supply and the circulating supply of the token.

Findings Breakdown

Critical	4
Medium	0
Minor / Informative	13

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	4	0	0	0
Medium	0	0	0	0
Minor / Informative	13	0	0	0

MT - Mints Tokens

Criticality	Critical
Location	contracts/Ploutos.sol#L127
Status	Unresolved

Description

The contract owner and the admin have the authority to mint new tokens indirectly through the `giveAllocation` function. By allocating tokens to users, they can effectively increase the total supply of tokens. Users can then claim these allocations using the `claimAllocation` function, which mints the allocated tokens to their account.

```
function giveAllocation(address user, uint256 amount) external
{
        require(msg.sender == admin || msg.sender == owner(),
"ACCESS DENIED");

        Allocation memory newAllocation = Allocation({
            totalAmount: amount,
            claimedAmount: 0,
            nextClaimTime: block.timestamp
        });

        allocations[user].push(newAllocation);

        emit AllocationIncreased(user, amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner’s account and admin'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.

PTPM - Potential Token Price Manipulation

Criticality	Critical
Location	contracts/Ploutos.sol#L46
Status	Unresolved

Description

The airdrop function allows users to receive a fixed amount of tokens by paying a fixed fee. Since the fee is constant, users can potentially exploit this function to manipulate the token price by repeatedly buying tokens at the fixed rate set by the contract and selling them at a potentially higher price. This could lead to significant price manipulation and market instability.

```
function airdrop() external payable nonReentrant {
        require(msg.value == FEE, "Incorrect fee");
        require(allocations[msg.sender].length == 0, "NOT
ALLOWED");
        payable(feeReceiver).transfer(msg.value);

        Allocation memory newAllocation = Allocation({
            totalAmount: AIRDROP_AMOUNT,
            claimedAmount: IMMEDIATE_AIRDROP,
            nextClaimTime: block.timestamp + DAY30
        });

        allocations[msg.sender].push(newAllocation);
        _mint(msg.sender, IMMEDIATE_AIRDROP);

        emit AirdropClaimed(msg.sender, AIRDROP_AMOUNT);
        emit AllocationClaimed(msg.sender, IMMEDIATE_AIRDROP);
    }
```

Recommendation

To mitigate this risk, consider implementing a pause functionality that allows the contract owner or admin to pause the airdrop function during periods of high volatility or suspected manipulation or after a period of time. This pause mechanism would enable better control over the airdrop process and help maintain market stability. Additionally, consider implementing a dynamic pricing mechanism for the airdrop fee that adjusts based on market conditions to further prevent price manipulation. This approach ensures that the airdrop functionality remains fair and does not adversely affect the token’s market price.

TCI - Token Cap Issue

Criticality	Critical
Location	contracts/Ploutos.sol#L58,79,108
Status	Unresolved

Description

The contract uses OpenZeppelin's ERC20Capped to enforce a maximum supply of tokens. The `maxSupply` is initialized to the specified cap, and tokens are minted to the `feeReceiver` at contract deployment. Additional tokens are minted to users during airdrop, presale purchases, and allocation claims. However, if the token cap is reached, no new tokens can be minted. This poses a significant issue as users who have bought tokens in the presale or through airdrop or have received allocations may not be able to claim their full entitlement if the cap is reached. For instance, users receive an immediate smaller amount of tokens (1% of the total bought) during the presale and are supposed to claim 1% more each month. If the cap is reached, these users will be unable to claim their remaining allocations, disrupting the intended token distribution.

```
function claimAllocation(uint index) external nonReentrant {
        require(index < allocations[msg.sender].length,
"Invalid index");
        Allocation storage allocation =
allocations[msg.sender][index];
        require(
            block.timestamp >= allocation.nextClaimTime,
            "Claim not yet available"
        );

        uint256 periodsElapsed = 1 +
            ((block.timestamp - allocation.nextClaimTime) /
DAY30);
        if (periodsElapsed > 0) {
            // Calculate the claimable amount based on the
periods elapsed
            uint256 claimable = (allocation.totalAmount *
periodsElapsed) / 100;
            if (
                claimable > (allocation.totalAmount -
allocation.claimedAmount)
            ) {
                claimable = allocation.totalAmount -
allocation.claimedAmount;
            }
            require(claimable > 0, "No claimable amount");

            allocation.claimedAmount += claimable;

            // Update the next claim time by adding the elapsed
time (periods * DAY30)
            allocation.nextClaimTime += periodsElapsed * DAY30;

            _mint(msg.sender, claimable);
            emit AllocationClaimed(msg.sender, claimable);
        } else {
            revert("No elapsed periods");
        }
    }
```

Recommendation

Consider implementing a mechanism to reserve a portion of the token supply specifically for future claims and allocations to ensure that users can claim their tokens over time without hitting the cap prematurely. One possible approach is to introduce a global variable that tracks the total amount of tokens allocated to users. This variable would ensure that the sum of the current total supply and the total amount allocated does not exceed the max supply. By doing so, you can prevent situations where users are unable to mint their tokens due to the cap being reached. Ensuring that the token distribution mechanism can function as intended without being disrupted by the cap is essential for maintaining user trust and the integrity of the presale and allocation processes.

TSD - Total Supply Diversion

Criticality	Critical
Location	contracts/Ploutos.sol#L166
Status	Unresolved

Description

The total supply of a token is the total number of tokens that have been created, while the balances of individual accounts represent the number of tokens that an account owns. The total supply and the balances of individual accounts are two separate concepts that are managed by different variables in a smart contract. These two entities should be equal to each other.In the contract, the `totalSupply` function is set to always return the maximum supply, regardless of the actual number of tokens minted and in circulation. This results in the total supply not accurately reflecting the actual supply of tokens. Consequently, the sum of the balances of individual accounts can diverge from the reported total supply.

```
function totalSupply() public pure override returns (uint256) {
    return maxSupply;
}
```

Recommendation

The total supply and the balance variables should be consistent and accurately represent the token distribution. The total supply represents the total number of tokens that have been created, while the balance mapping stores the number of tokens that each account owns. Modify the `totalSupply` function to ensure that it reflects the actual number of tokens in circulation, so the sum of balances always equals the total supply. This adjustment is essential for maintaining accurate and transparent accounting of the token's supply.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L115,121,127
Status	Unresolved

Description

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

```
function setPresaleRate(uint256 _rate) external {
        require(msg.sender == admin || msg.sender == owner(),
"ACCESS DENIED");
        presaleRate = _rate;
        emit PresaleRateChanged(_rate);
    }

    function startStopPresale(bool _status) external {
        require(msg.sender == admin || msg.sender == owner(),
"ACCESS DENIED");
        presaleActive = _status;
        emit PresaleStatusChanged(_status);
    }

    function giveAllocation(address user, uint256 amount)
external {
        require(msg.sender == admin || msg.sender == owner(),
"ACCESS DENIED");

        Allocation memory newAllocation = Allocation({
            totalAmount: amount,
            claimedAmount: 0,
            nextClaimTime: block.timestamp
        });

        allocations[user].push(newAllocation);

        emit AllocationIncreased(user, amount);
    }
```

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.

IAC - Immediate Allocation Claim

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L83,127
Status	Unresolved

Description

The `giveAllocation` function allows the owner to allocate tokens to users, setting the `nextClaimTime` to the current block timestamp. This enables users to claim their allocations immediately using the `claimAllocation` function. The logic in the `claimAllocation` function includes a calculation for `periodsElapsed` that starts at 1, effectively treating the allocation as if one month has already passed. This discrepancy allows users to claim their allocation without waiting for the intended one-month period, leading to premature token distribution.

```
function claimAllocation(uint index) external nonReentrant {
        require(index < allocations[msg.sender].length,
"Invalid index");
        Allocation storage allocation =
allocations[msg.sender][index];
        require(
            block.timestamp >= allocation.nextClaimTime,
            "Claim not yet available"
        );

        uint256 periodsElapsed = 1 +
            ((block.timestamp - allocation.nextClaimTime) /
DAY30);
        if (periodsElapsed > 0) {
            // Calculate the claimable amount based on the
periods elapsed
            uint256 claimable = (allocation.totalAmount *
periodsElapsed) / 100;
            if (
                claimable > (allocation.totalAmount -
allocation.claimedAmount)
            ) {
                claimable = allocation.totalAmount -
allocation.claimedAmount;
            }
            require(claimable > 0, "No claimable amount");

            allocation.claimedAmount += claimable;

            // Update the next claim time by adding the elapsed
time (periods * DAY30)
            allocation.nextClaimTime += periodsElapsed * DAY30;

            _mint(msg.sender, claimable);
            emit AllocationClaimed(msg.sender, claimable);
        } else {
            revert("No elapsed periods");
        }
    }

function giveAllocation(address user, uint256 amount) external
{
        require(msg.sender == admin || msg.sender == owner(),
"ACCESS DENIED");

        Allocation memory newAllocation = Allocation({
            totalAmount: amount,
            claimedAmount: 0,
            nextClaimTime: block.timestamp
        });
```

```
        allocations[user].push(newAllocation);

        emit AllocationIncreased(user, amount);
    }
```

Recommendation

To align the claiming mechanism with the intended vesting period, the `nextClaimTime` in the `giveAllocation` function should be set to a future date, ensuring users must wait for the specified period before claiming their tokens. Additionally, review and adjust the logic in the `claimAllocation` function to accurately reflect the elapsed time and prevent immediate claims. Ensuring consistent and logical allocation and claiming mechanisms is crucial for maintaining the intended token distribution schedule and preventing premature token access.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L41,42
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.

```
admin
feeReceiver
```

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.

MC - Misleading Comment

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L43
Status	Unresolved

Description

The constructor of the contract includes a comment indicating that tokens are minted to the `admin` address. However, the implementation actually mints tokens to the `feeReceiver` address. This discrepancy between the comment and the actual behavior of the code can lead to confusion and misunderstandings about the initial token distribution. Accurate comments are crucial for maintaining clarity.

```
constructor(
        address _admin,
        address _feeReceiver
    ) ERC20("PLOUTOS", "PLTL") ERC20Capped(maxSupply)
Ownable(msg.sender) {
        require(_feeReceiver != address(0), "Invalid fee
receiver address");
        admin = _admin;
        feeReceiver = _feeReceiver;
        _mint(_feeReceiver, 11000046153 * 10 ** 7); // Mint
110000461.53 tokens to admin
    }
```

Recommendation

The comment in the constructor should be corrected to accurately reflect that the tokens are minted to the feeReceiver address, not the admin address. Ensuring that comments accurately describe the code's functionality helps maintain clear documentation and prevents potential misunderstandings or misinterpretations of the contract's behavior. Clear and precise comments are essential for the proper maintenance of the smart contract.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L116,122,128
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(msg.sender == admin || msg.sender == owner(), "ACCESS
DENIED");
```

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.

NMU - NonReentrant Modifier usage

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L46,64,83
Status	Unresolved

Description

The `airdrop` and `buyPresale` functions in the smart contract utilize the `nonReentrant` modifier to prevent reentrancy attacks. These functions make external calls to transfer Ether to the `feeReceiver` and `admin` addresses, respectively. While the `nonReentrant` modifier is justified to prevent potential reentrancy issues during these transfers, it is important to note that the addresses receiving Ether are set by the contract owner. The `claimAllocation` function also uses the `nonReentrant` modifier, even though it does not make any external calls or Ether transfers. This function interacts solely with the contract's internal state and mints new tokens. The use of the `nonReentrant` modifier here is unnecessary.

```
function airdrop() external payable nonReentrant {
        require(msg.value == FEE, "Incorrect fee");
        require(allocations[msg.sender].length == 0, "NOT
ALLOWED");
        payable(feeReceiver).transfer(msg.value);

        Allocation memory newAllocation = Allocation({
            totalAmount: AIRDROP_AMOUNT,
            claimedAmount: IMMEDIATE_AIRDROP,
            nextClaimTime: block.timestamp + DAY30
        });

        allocations[msg.sender].push(newAllocation);
        _mint(msg.sender, IMMEDIATE_AIRDROP);

        emit AirdropClaimed(msg.sender, AIRDROP_AMOUNT);
        emit AllocationClaimed(msg.sender, IMMEDIATE_AIRDROP);
    }

    function buyPresale() external payable nonReentrant {
        require(presaleActive, "Presale is not active");
        require(presaleRate > 0, "Presale is not set");
        uint256 amount = (msg.value * presaleRate) / (1 ether);
        uint256 immediateAmount = amount / 100;
        payable(admin).transfer(msg.value);

        allocations[msg.sender].push(
            Allocation({
                totalAmount: amount,
                claimedAmount: immediateAmount,
                nextClaimTime: block.timestamp + DAY30
            })
        );

        _mint(msg.sender, immediateAmount);
        emit PresalePurchased(msg.sender, amount);
    }

    function claimAllocation(uint index) external nonReentrant
{
        require(index < allocations[msg.sender].length,
"Invalid index");
        Allocation storage allocation =
allocations[msg.sender][index];
        require(
            block.timestamp >= allocation.nextClaimTime,
            "Claim not yet available"
        );

        uint256 periodsElapsed = 1 +
```

```
            ((block.timestamp - allocation.nextClaimTime) /
DAY30);
        if (periodsElapsed > 0) {
            // Calculate the claimable amount based on the
periods elapsed
            uint256 claimable = (allocation.totalAmount *
periodsElapsed) / 100;
            if (
                claimable > (allocation.totalAmount -
allocation.claimedAmount)
            ) {
                claimable = allocation.totalAmount -
allocation.claimedAmount;
            }
            require(claimable > 0, "No claimable amount");

            allocation.claimedAmount += claimable;

            // Update the next claim time by adding the elapsed
time (periods * DAY30)
            allocation.nextClaimTime += periodsElapsed * DAY30;

            _mint(msg.sender, claimable);
            emit AllocationClaimed(msg.sender, claimable);
        } else {
            revert("No elapsed periods");
        }
    }
```

Recommendation

Evaluate the necessity of the `nonReentrant` modifier in the `airdrop`, `buyPresale`, since the `airdrop` and `buyPresale` functions involve transferring Ether to addresses controlled by the contract owner. For the `claimAllocation` function, assess whether the `nonReentrant` modifier is required given that it does not involve external calls. Removing the modifier could simplify the code without compromising security.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L46,64
Status	Unresolved

Description

The contract sends funds to a `feeReceiver` and `admin` as part of the `airdrop` and `buyPresale` flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
function airdrop() external payable nonReentrant {
        require(msg.value == FEE, "Incorrect fee");
        require(allocations[msg.sender].length == 0, "NOT
ALLOWED");
        payable(feeReceiver).transfer(msg.value);

        Allocation memory newAllocation = Allocation({
            totalAmount: AIRDROP_AMOUNT,
            claimedAmount: IMMEDIATE_AIRDROP,
            nextClaimTime: block.timestamp + DAY30
        });

        allocations[msg.sender].push(newAllocation);
        _mint(msg.sender, IMMEDIATE_AIRDROP);

        emit AirdropClaimed(msg.sender, AIRDROP_AMOUNT);
        emit AllocationClaimed(msg.sender, IMMEDIATE_AIRDROP);
    }

    function buyPresale() external payable nonReentrant {
        require(presaleActive, "Presale is not active");
        require(presaleRate > 0, "Presale is not set");
        uint256 amount = (msg.value * presaleRate) / (1 ether);
        uint256 immediateAmount = amount / 100;
        payable(admin).transfer(msg.value);

        allocations[msg.sender].push(
            Allocation({
                totalAmount: amount,
                claimedAmount: immediateAmount,
                nextClaimTime: block.timestamp + DAY30
            })
        );

        _mint(msg.sender, immediateAmount);
        emit PresalePurchased(msg.sender, amount);
    }
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

PSI - Presale Status Initialization

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L22,121
Status	Unresolved

Description

The `presaleActive` boolean variable is initialized to `true`, which means the presale functionality is enabled by default upon deployment. This could lead to unintended presale activity before the contract owner has had the opportunity to verify that all presale parameters, including the `presaleRate`, are correctly set.

```
bool public presaleActive = true;

function startStopPresale(bool _status) external {
        require(msg.sender == admin || msg.sender == owner(),
"ACCESS DENIED");
        presaleActive = _status;
        emit PresaleStatusChanged(_status);
    }
```

`

Recommendation

It is recommended to initialize the `presaleActive` variable to `false`. The owner should then explicitly activate the presale by calling the startStopPresale function once they are certain that all presale parameters have been correctly configured. This practice enhances security by ensuring that the presale does not start prematurely and that all necessary conditions are met before activation. Additionally, maintaining a manual activation process for the presale provides an extra layer of control and verification.

RC - Redundant Check

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L83
Status	Unresolved

Description

The `claimAllocation` function includes a calculation for `periodsElapsed` that starts from 1. Following this, there is a check to ensure that `periodsElapsed` is greater than 0, which is redundant because `periodsElapsed` will always be at least 1. This unnecessary check adds clutter to the code and does not contribute to the logic or security of the function.

```
function claimAllocation(uint index) external nonReentrant {
        require(index < allocations[msg.sender].length,
"Invalid index");
        Allocation storage allocation =
allocations[msg.sender][index];
        require(
            block.timestamp >= allocation.nextClaimTime,
            "Claim not yet available"
        );

        uint256 periodsElapsed = 1 +
            ((block.timestamp - allocation.nextClaimTime) /
DAY30);
        if (periodsElapsed > 0) {
            // Calculate the claimable amount based on the
periods elapsed
            uint256 claimable = (allocation.totalAmount *
periodsElapsed) / 100;
            if (
                claimable > (allocation.totalAmount -
allocation.claimedAmount)
            ) {
                claimable = allocation.totalAmount -
allocation.claimedAmount;
            }
            require(claimable > 0, "No claimable amount");

            allocation.claimedAmount += claimable;

            // Update the next claim time by adding the elapsed
time (periods * DAY30)
            allocation.nextClaimTime += periodsElapsed * DAY30;

            _mint(msg.sender, claimable);
            emit AllocationClaimed(msg.sender, claimable);
        } else {
            revert("No elapsed periods");
        }
    }
```

Recommendation

Remove the redundant check for `periodsElapsed` being greater than 0, as it will always be true given the current calculation. Streamlining the function by eliminating superfluous conditions helps maintain cleaner and more readable code. It is essential to ensure that checks and conditions in the smart contract serve a necessary purpose and contribute to its intended functionality.

TNE - Typographical Naming Error

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L170
Status	Unresolved

Description

The function `circlatingSupply` is intended to return the circulating supply of tokens. However, there is a typographical error in the function name where it is written as `circlatingSupply` instead of circulatingSupply. This typo can lead to confusion and potential issues when interacting with the contract, as the function name does not accurately reflect its intended purpose.

```
function circlatingSupply() public view returns (uint256) {
        return super.totalSupply();
    }
```

Recommendation

The function name should be corrected to `circulatingSupply` to accurately describe its functionality. This change will improve the readability and maintainability of the code, ensuring that the function's purpose is clear to users interacting with the contract. Clear and accurate naming conventions are crucial for the effective understanding and management of smart contracts.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Ploutos.sol#L115,121
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.

```
uint256 _rate
bool _status
```

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	contracts/Ploutos.sol#L41
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.

```
admin = _admin
```

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	contracts/Ploutos.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 ^0.8.24;
```

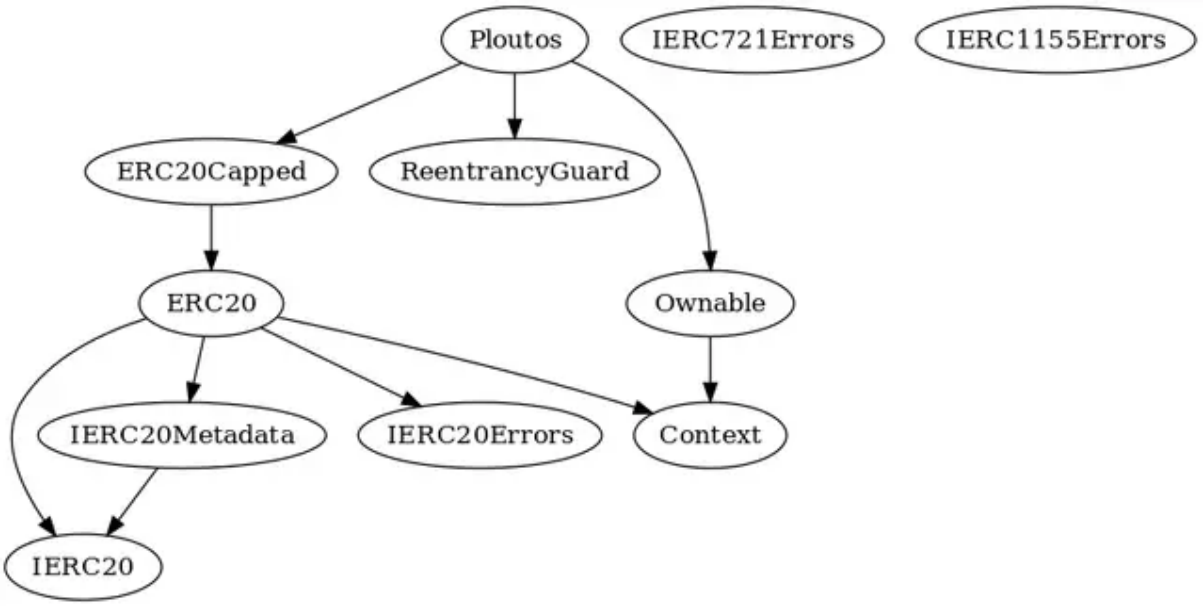
Recommendation

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.

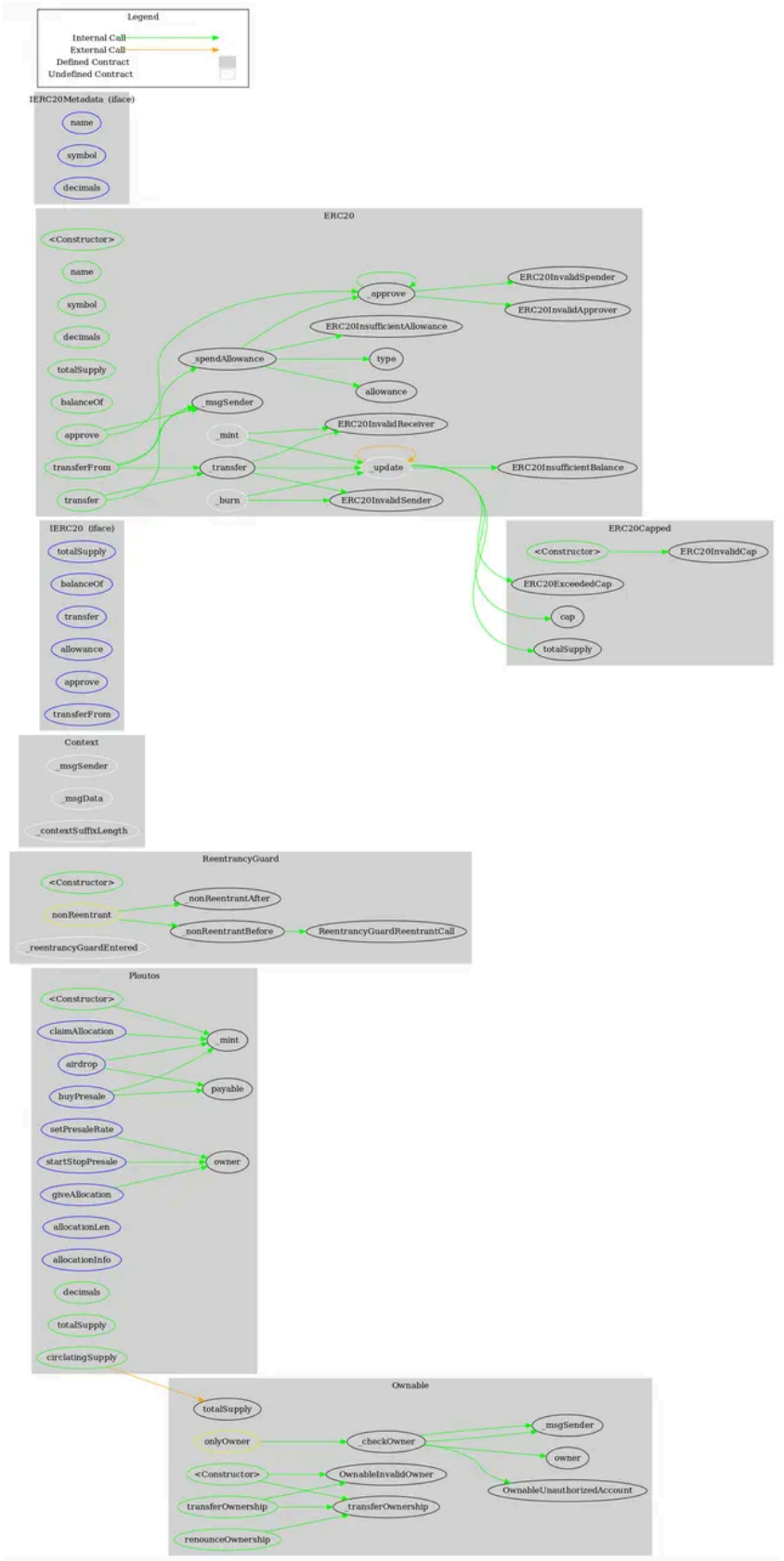
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Ploutos	Implementation	ERC20Capped, ReentrancyGuard, Ownable		
		Public	✓	ERC20 ERC20Capped Ownable
	airdrop	External	Payable	nonReentrant
	buyPresale	External	Payable	nonReentrant
	claimAllocation	External	✓	nonReentrant
	setPresaleRate	External	✓	-
	startStopPresale	External	✓	-
	giveAllocation	External	✓	-
	allocationLen	External		-
	allocationInfo	External		-
	decimals	Public		-
	totalSupply	Public		-
	circlatingSupply	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IERC20Metadata (iface)
name
symbol
decimals
ERC20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
approve
transferFrom
transfer
_spendAllowance
_msgSender
_mint
transfer
_burn
_approve
ERC20InsufficientAllowance
type
allowance
ERC20InvalidReceiver
_update
ERC20InvalidSpender
ERC20InvalidApprover
ERC20InsufficientBalance
ERC20InvalidSender
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
ERC20Capped
<Constructor>
ERC20InvalidCap
ERC20ExceededCap
cap
totalSupply
Context
_msgSender
_msgData
_contextSuffixLength
ReentrancyGuard
<Constructor>
nonReentrant
_reentrancyGuardEntered
_nonReentrantAfter
_nonReentrantBefore
ReentrancyGuardReentrantCall
Ploutos
<Constructor>
claimAllocation
airdrop
buyPresale
setPresaleRate
startStopPresale
giveAllocation
allocationLen
allocationInfo
decimals
totalSupply
circlatingSupply
_mint
payable
owner
Ownable
totalSupply
onlyOwner
<Constructor>
transferOwnership
renounceOwnership
_checkOwner
OwnableInvalidOwner
_transferOwnership
_msgSender
owner
OwnableUnauthorizedAccount

Summary

PLOUTOS contract implements a token and ido mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like mint tokens. if the contract owner abuses the mint functionality, then the contract will be highly inflated. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats.

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