Cyberscope

# Audit Report

# PLOUTOS

September 2024

| | |
|---|---|
| Network | Base |
| Address | 0x236f9f7437333cBB27fA7E7565049B79d174ea04 |
| 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 | Passed |
| ● | BT | Burns Tokens | Passed |
| ● | BC | Blacklists Addresses | Passed |

# Diagnostics

Critical Medium Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| | CCR | Contract Centralization Risk | Unresolved |
| | IDI | Immutable Declaration Improvement | Unresolved |
| | NMU | NonReentrant Modifier usage | Unresolved |
| | PTRP | Potential Transfer Revert Propagation | Unresolved |
| | PSI | Presale Status Initialization | Unresolved |
| | RC | Redundant Check | Unresolved |
| | L04 | Conformance to Solidity Naming Conventions | Unresolved |
| | L16 | Validate Variable Setters | Unresolved |
| | L19 | Stable Compiler Version | Unresolved |
| | L20 | Succeeded Transfer Check | Unresolved |

# Table of Contents

# Review

| | |
|---|---|
| **Contract Name** | Ploutos |
| **Compiler Version** | v0.8.24+commit.e11b9ed9 |
| **Optimization** | 200 runs |
| **Testing Deploy** | https://testnet.bscscan.com/address/0x92c0fdbaf4d142e85fb1e9740ca88f5454df7eaa |
| **Explorer** | https://basescan.org/address/0x236f9f7437333cbb27fa7e7565049b79d174ea04 |
| **Address** | 0x236f9f7437333cbb27fa7e7565049b79d174ea04 |
| **Network** | BASE |
| **Symbol** | PLTL |
| **Decimals** | 9 |
| **Total Supply** | 210,000,461.53 |
| **Badge Eligibility** | Yes |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 03 Jul 2024 |
| **Corrected Phase 2** | 05 Sep 2024 |

## Source Files

| Filename | SHA256 |
|---|---|
| **contracts/Ploutos.sol** | 14c89fcc385c1d6430b13ac1db466571e10a1c31b000e1f791290047b468d750 |

| | |
|---|---|
| **@openzeppelin/contracts/utils/ReentrancyGuard.sol** | 8d0bac508a25133c9ff80206f65164cef959ec084645d1e7b06050c2971ae0fc |
| **@openzeppelin/contracts/utils/Context.sol** | 9c1cc43aa4a2bde5c7dea0d4830cd42c54813ff883e55c8d8f12e6189bf7f10a |
| **@openzeppelin/contracts/token/ERC20/IERC20.sol** | 6f2faae462e286e24e091d7718575179644dc60e79936ef0c92e2d1ab3ca3cee |
| **@openzeppelin/contracts/token/ERC20/ERC20.sol** | 2d874da1c1478ed22a2d30dcf1a6ec0d09a13f897ca680d55fb49fbcc0e0c5b1 |
| **@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:

**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.

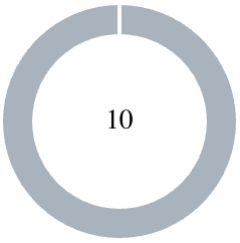
**Events and Emissions**

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 | 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 |

# CCR - Contract Centralization Risk

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Ploutos.sol#L124,129,134 |
| **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) isAdministrator
external {
        presaleRate = _rate;
        emit PresaleRateChanged(_rate);
    }

    function startStopPrivateSale(bool _status) isAdministrator
external {
        presaleActive = _status;
        emit PresaleStatusChanged(_status);
    }

    function giveAllocation(address user, uint256 amount)
isAdministrator external {
    ...
    }
```

## 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.

## IDI - Immutable Declaration Improvement

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Ploutos.sol#L60 |
| **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
```

### 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.

## NMU - NonReentrant Modifier usage

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

## Description

The `buyPrivateSale` and `claimAllocation` functions in the smart contract utilize the `nonReentrant` modifier to prevent reentrancy attacks. These functions make external calls to transfer Ether to the and `admin` addresses. While the `nonReentrant` modifier is justified to prevent potential reentrancy issues during the transfers, it is important to note that the address receiving Ether is 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 transfer tokens. As a result the use of the `nonReentrant` modifier here is unnecessary.

```
    function buyPrivateSale() external payable nonReentrant {
        require(presaleActive, "Private sale is not active");
        require(presaleRate > 0, "Private sale is not set");
        uint256 amount = (msg.value * presaleRate) / (1 ether);
        uint256 immediateAmount = amount / 100;

        require(
            token.balanceOf(address(this)) >=
                unclaimedAllocation + amount,
            "NOT ENOUGH TOKEN IN DISTRIBUTOR"
        );

        payable(admin).transfer(msg.value);

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

        unclaimedAllocation += (amount - immediateAmount);

        token.transfer(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;

            token.transfer(msg.sender, claimable);
            unclaimedAllocation -= claimable;
            emit AllocationClaimed(msg.sender, claimable);
        } else {
            revert("No elapsed periods");
        }
    }
```

## Recommendation

Evaluate the necessity of the `nonReentrant` modifier in the `buyPrivateSale`, `claimAllocation`, since these 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#L75 |
| **Status** | Unresolved |

### Description

The contract sends funds to an `admin` as part of the `buyPrivateSale` 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 buyPrivateSale() external payable nonReentrant {
        ...

        payable(admin).transfer(msg.value);

        ...
```

### 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#L39,129 |
| **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 startStopPrivateSale(bool _status) isAdministrator
external {
        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#L99 |
| **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) {
        ...
        }
```

## 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.

## L04 - Conformance to Solidity Naming Conventions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Ploutos.sol#L124,129 |
| **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#L60 |
| **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.

## L20 - Succeeded Transfer Check

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Ploutos.sol#L87,116 |
| **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.

```
token.transfer(msg.sender, immediateAmount)
token.transfer(msg.sender, claimable)
```

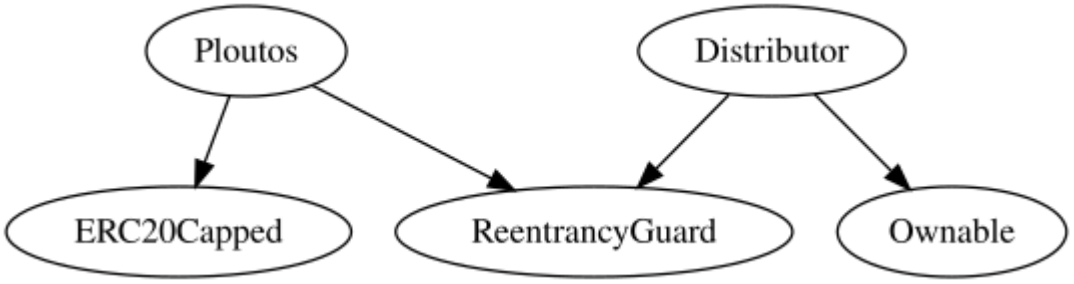
### Recommendation

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.

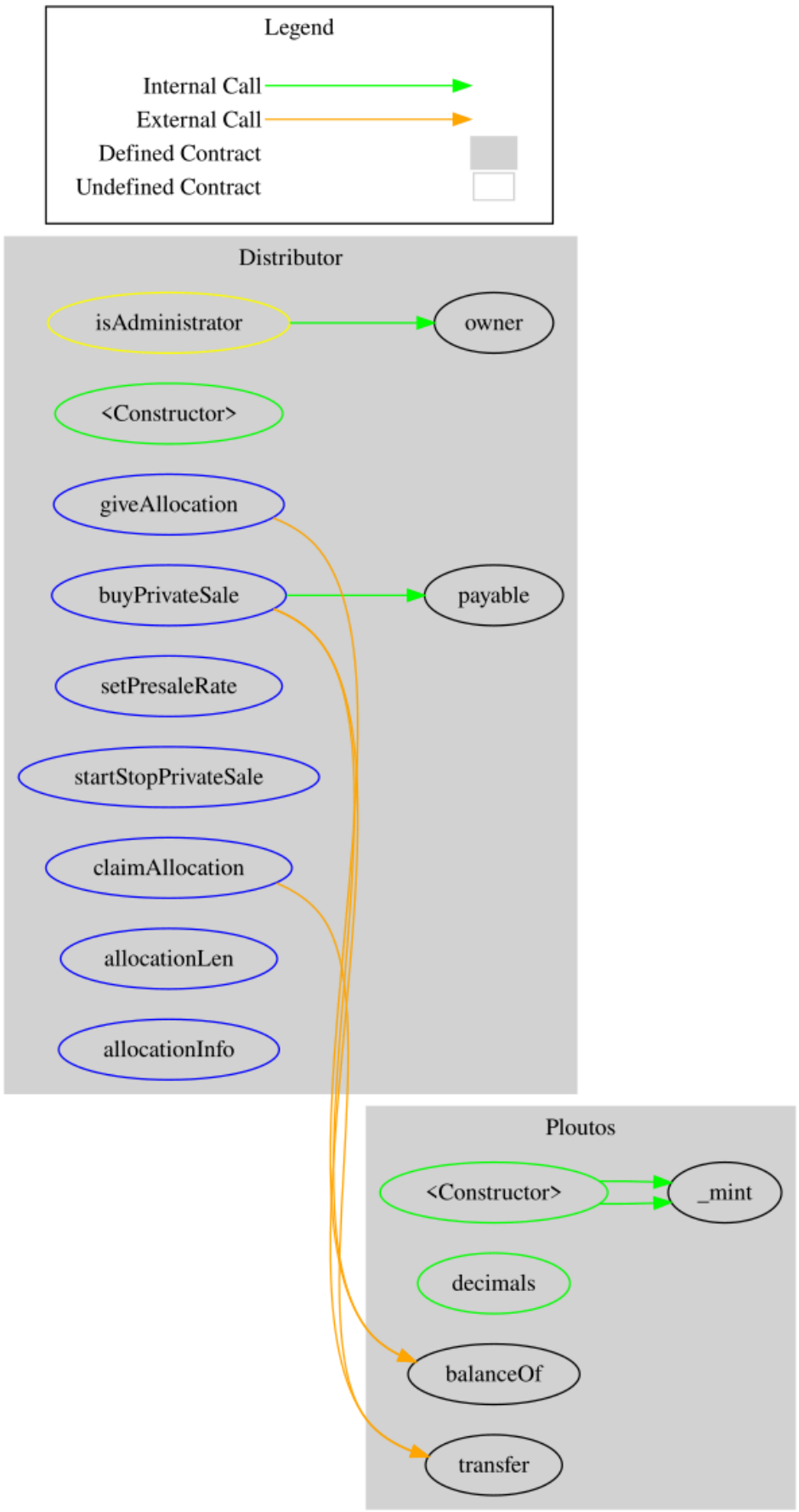
# Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | **Function Name** | **Visibility** | **Mutability** | **Modifiers** |
| | | | | |
| **Ploutos** | Implementation | ERC20Capped, ReentrancyGuard | | |
| | | Public | ✓ | ERC20 ERC20Capped |
| | decimals | Public | | - |
| | | | | |
| **Distributor** | Implementation | ReentrancyGuard, Ownable | | |
| | | Public | ✓ | Ownable |
| | buyPrivateSale | External | Payable | nonReentrant |
| | claimAllocation | External | ✓ | nonReentrant |
| | setPresaleRate | External | ✓ | isAdministrator |
| | startStopPrivateSale | External | ✓ | isAdministrator |
| | giveAllocation | External | ✓ | isAdministrator |
| | allocationLen | External | | - |
| | allocationInfo | External | | - |

# Inheritance Graph



# Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Distributor
isAdministrator
owner
<Constructor>
giveAllocation
buyPrivateSale
payable
setPresaleRate
startStopPrivateSale
claimAllocation
allocationLen
allocationInfo
Ploutos
<Constructor>
_mint
decimals
balanceOf
transfer

# Summary

PLOUTOS contract implements a token and ido mechanism. This audit investigates security issues, business logic concerns and potential improvements. PLOUTOS is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions.

# 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