Cyberscope

A TAC Security Company

Audit Report
Lyno AI

July 2025

Repository	https://github.com/lyno-ai/lyno_presale
Commit	9aea015fa862fbdfee823c1874510eaab5832970
Files	LynoPresale.sol
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/lyno-ai/lyno_presale
Commit	9aea015fa862fbdfee823c1874510eaab5832970

Audit Updates

Initial Audit	03 Jul 2025

Source Files

Filename	SHA256
LynoPresale.sol	a93f99f26b415678e00af3fbc5cc503496497d7159112ef284e8b8852ace be21

Overview

The `LynoPresale` contract manages a multi-stage token presale for the `Lyno` token, supporting contributions in ETH, USDC, and USDT. It enables token purchases across defined stages, each with specific pricing, bonus rates, and token allocations. The contract incorporates security features like reentrancy protection, pausability, and ownership controls. It also facilitates token claiming post-sale and supports Chainlink price feeds for real-time ETH/USD conversion.

Multi-Stage Token Sale Functionality

The contract defines up to seven presale stages with the following customizable parameters per stage:

- **Stage Name**
- **Token Price (USD with 18 decimals)**
- **Bonus Percentage**
- **Token Allocation**
- **Tokens Sold**
- **Active Status**

Only one stage can be active at a time. The owner can change stages using the `setStage` function, ensuring orderly progression through the sale. The `setupDefaultStages` function initializes the stages with predetermined values that follow the community distribution plan.

Purchase Token Functionality

Buyers can participate in the presale using one of the three accepted currencies:

- **ETH**: The contract uses Chainlink's ETH/USD price feed to convert ETH to USD.
- **USDC / USDT**: Tokens are transferred using `SafeERC20` and normalized to 18 decimals.

Upon contribution:

- The token amount is calculated based on the current stage's price.
- A bonus is applied according to the stage's bonus percentage.
- Token purchase stats are updated.
- ETH is immediately forwarded to the `ethReceiver` wallet.
- Contributions are tracked per user and token.

The `buyWithETH`, `buyWithUSDC`, and `buyWithUSDT` functions each handle their respective token logic securely and efficiently.

Token Claiming Functionality

The owner can enable token claiming using `enableClaiming`. Users who contributed can then call `claim` to receive their purchased tokens. The function ensures:

- Claiming is enabled.
- The user has tokens to claim.
- The user has not already claimed.

Tokens are distributed using `SafeERC20`, and each claim is recorded to prevent double-claims.

Information Retrieval Functionality

The contract provides user-friendly views and stats to track presale progress:

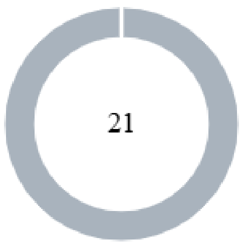
- `getCurrentStage`: Returns details of the active stage.
- `getContribution(user, token)`: Shows user's contribution in ETH, USDC, or USDT.
- `totalContributions(token)`: Displays the total contributions for a given token.
- `getEthPrice`: Returns the latest ETH/USD price using Chainlink's oracle.

These functions are critical for building responsive and transparent frontend interfaces.

Administrative and Safety Features

- **Ownership Control**: Only the contract owner can:
  - Set the Lyno token address
  - Change presale stages
  - Enable claiming
  - Withdraw collected USDC/USDT tokens
- **Pausing and Unpausing**: The `pause` and `unpause` functions can disable or resume purchases during emergencies.
- **Reentrancy Protection**: All sensitive external functions (e.g., `buyWithETH`, `claim`) use the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard`.
- **Fund Safety**: USDC and USDT funds are securely held and can be withdrawn only by the owner using `withdrawERC20`.

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
●	MVN	Misleading Variables Naming	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	HV	Hardcoded Values	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IPC	Inconsistent Phase Change	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	PPF	Pausable Purchase Functionality	Unresolved
●	PECC	Potential Early Claim Concern	Unresolved
●	POSD	Potential Oracle Stale Data	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	PPA	Pre-Configuration Purchase Allowance	Unresolved

●	RAO	Redundant Arithmetic Operations	Unresolved
●	RSP	Redundant Struct Property	Unresolved
●	TCEC	Token Claiming Ensurance Concern	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	UTPD	Unverified Third Party Dependencies	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	LynoPresale.sol#L21
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

```
address public constant ETH_ADDRESS =
address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE);
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

CR - Code Repetition

Criticality	Minor / Informative
Location	LynoPresale.sol#L212,247,280
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 buyWithETH() external payable nonReentrant
whenNotPaused
function buyWithUSDC(uint256 amount) external nonReentrant
whenNotPaused
function buyWithUSDT(uint256 amount) external nonReentrant
whenNotPaused
```

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	LynoPresale.sol#L163,172,182,330,369,376
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 setToken(address _lynoToken) external onlyOwner
function enableClaiming() external onlyOwner
function setStage(uint8 newStageId) external onlyOwner
function withdrawERC20(address token) external onlyOwner
function pause() external onlyOwner
function unpause() external onlyOwner
```

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.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	LynoPresale.sol#L321
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.

```
function claim() external nonReentrant {
    //...
    uint256 amount = purchasedTokens[msg.sender];
    //...
    IERC20(lynoToken).safeTransfer(msg.sender, amount);
    //...
}
```

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.

HV - Hardcoded Values

Criticality	Minor / Informative
Location	LynoPresale.sol#L183,206,253,286
Status	Unresolved

Description

The contract contains multiple instances where numeric values are directly hardcoded into the code logic rather than being assigned to constant variables with descriptive names. Hardcoding such values can lead to several issues, including reduced code readability, increased risk of errors during updates or maintenance, and difficulty in consistently managing values throughout the contract. Hardcoded values can obscure the intent behind the numbers, making it challenging for developers to modify or for users to understand the contract effectively.

```
require(newStageId <= 6, "Invalid stage ID");
return uint256(price) * 10**10;
uint256 normalizedAmount = amount * 10**12;
```

Recommendation

It is recommended to replace hardcoded numeric values with variables that have meaningful names. This practice improves code readability and maintainability by clearly indicating the purpose of each value, reducing the likelihood of errors during future modifications. Additionally, consider using constant variables which provide a reliable way to centralize and manage values, improving gas optimization throughout the contract.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	LynoPresale.sol#L79,80,81
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.

```
usdcAddress
usdtAddress
ethReceiver
```

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.

IPC - Inconsistent Phase Change

Criticality	Minor / Informative
Location	LynoPresale.sol#L182
Status	Unresolved

Description

The contract has the function `setStage` allowing only the owner to set the new stage of presale. However there are several inconsistencies of how the contract is handling the setting of stage.

Specifically:

- The stage can be set without following the intended order.
- The stage can be changed to a previous stage that has already been completed.
- The stage can be changed without being complete.
- If a stage is completed this function must be called otherwise the user will not be able to buy tokens.

```
function setStage(uint8 newStageId) external onlyOwner {
    require(newStageId <= 6, "Invalid stage ID");
    require(newStageId != currentStageId, "Already in this
stage");
    stages[currentStageId].active = false;
    stages[newStageId].active = true;
    emit StageChanged(currentStageId, newStageId);
    currentStageId = newStageId;
}
```

Recommendation

The team is advised to use a more automated and in sequence approach to handle the change of phase. This could be achieved by handling the phase change duing the purchase of the tokens with the proper conditions to ensure that operations run as intended.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	LynoPresale.sol#L163,330
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 setToken(address _lynoToken) external onlyOwner
function withdrawERC20(address token) external onlyOwner
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	LynoPresale.sol#L206
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 getEthPrice() public view returns (uint256) {
    (, int256 price, , , ) = ethPriceFeed.latestRoundData();
    //...
    return uint256(price) * 10**10;
}
```

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.

PPF - Pausable Purchase Functionality

Criticality	Minor / Informative
Location	LynoPresale.sol#L212,247,280,369
Status	Unresolved

Description

The contract owner is able to pause the purchasing of tokens by using the pause function. This will not allow users to purchase tokens. Additionally the owner is able to pause the purchases by setting the current stage in a complete one as described in the `IPC` section.

```
function buyWithETH() external payable nonReentrant
whenNotPaused
...
function buyWithUSDC(uint256 amount) external nonReentrant
whenNotPaused
...
function buyWithUSDT(uint256 amount) external nonReentrant
whenNotPaused
...
function pause() external onlyOwner {
    _pause();
}
```

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.

PECC - Potential Early Claim Concern

Criticality	Minor / Informative
Location	LynoPresale.sol#L172,312
Status	Unresolved

Description

The contract allows the owner to use the `enableClaiming` function allowing users to claim their purchased tokens. However it is possible that the function is triggered before the end of the presale and at any phase. This creates potential inconsistencies in the normal order of the contract's operations.

Specifically:

- The contract will hold unsold tokens while the users will be able to claim the purchased ones.
- Users that claim their token can still purchase more without a way to receive them since they can only claim tokens once. This may probably result in unrecoverable tokens
- Users may use their purchased tokens to create pools of undesirable liquidity ratios in decentralized exchanges before the presale ends.

```
function enableClaiming() external onlyOwner {
    require(lynoToken != address(0), "Token not set");
    claimingEnabled = true;
    emit ClaimingEnabled();
}
...
function claim() external nonReentrant {
    //...
    require(!hasClaimed[msg.sender], "Already claimed");
    //...
    IERC20(lynoToken).safeTransfer(msg.sender, amount);
}
```

Recommendation

It is recommended that the team creates the mechanisms needed to ensure that the claiming of tokens can only happen after the ending of the presale. This can be achieved by implementing a time mechanism to ensure that the processes are in order.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	LynoPresale.sol#L202
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
function getEthPrice() public view returns (uint256) {
     (, int256 price, , , ) = ethPriceFeed.latestRoundData();
     //...
}
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	LynoPresale.sol#L271,304
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
IERC20(usdcAddress).safeTransferFrom(msg.sender, address(this),
amount);
...
IERC20(usdtAddress).safeTransferFrom(msg.sender, address(this),
amount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
 Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

PPA - Pre-Configuration Purchase Allowance

Criticality	Minor / Informative
Location	LynoPresale.sol#L163,212,247,280
Status	Unresolved

Description

The contract allows users to purchase tokens before critical configurations are set. Specifically, the `setToken` function can be used after users use the buy functions. While this does not cause any inconsistencies in the buy functions, it may erode user trust by allowing purchases before critical configurations are in place.

```
function setToken(address _lynoToken) external onlyOwner {
    require(_lynoToken != address(0), "Invalid token address");
    require(lynoToken == address(0), "Token already set");
    lynoToken = _lynoToken;
}

function buyWithETH() external payable nonReentrant
whenNotPaused
function buyWithUSDC(uint256 amount) external nonReentrant
whenNotPaused
function buyWithUSDT(uint256 amount) external nonReentrant
whenNotPaused
```

Recommendation

The team could implement a check to ensure that critical configurations have been set before the users are allowed to purchase tokens.

RAO - Redundant Arithmetic Operations

Criticality	Minor / Informative
Location	LynoPresale.sol#L219,222
Status	Unresolved

Description

In `buyWithETH` function `ethInUSD` is calculated by multiplying the `msg.value` with the current eth price and then dividing with `PRECISION`. However in the next step, to calculate the `baseTokens` the function multiplies `ethInUSD` with `PRECISION` and then divides with the current stage's price. These redundant operation increase the code complexity and gas costs.

```
uint256 ethInUSD = (msg.value * ethPrice) / PRECISION;
uint256 baseTokens = (ethInUSD * PRECISION) / stage.price;
```

Recommendation

The team should restructure the function to ensure that there are no redundant calculations to enhance code optimization and efficiency.

RSP - Redundant Struct Property

Criticality	Minor / Informative
Location	LynoPresale.sol#L49,187,190
Status	Unresolved

Description

`Stage` struct has the property `active`. This property is used to check which is the current active stage. However, the pick of `stage` during purchase is handled by the `currentStageId` which is updated every time the stage changes, therefore the `active` property does not provide any additional value. Additionally, during the purchase functions the contract performs a check to ensure that `stage.active` is true. This check is also redundant since the current stage will always have the property `active` as true.

```
struct Stage {
    //...
    bool active;
}
function setStage(uint8 newStageId) external onlyOwner {
    //...
    stages[currentStageId].active = false;
    stages[newStageId].active = true;
    //...
    currentStageId = newStageId;
}
function buyWithETH() external payable nonReentrant
whenNotPaused {
    //...
    Stage storage stage = stages[currentStageId];
    require(stage.active, "Current stage not active");
    //...
}
```

Recommendation

The team is advised to remove redundancies to ensure that the code is optimised in terms of gas costs and readable.

TCEC - Token Claiming Ensurance Concern

Criticality	Minor / Informative
Location	LynoPresale.sol#L172,237,315,330
Status	Unresolved

Description

The contract does not ensure that users will be able to claim their purchased tokens. This is due to the enabling of claiming not being an automated action and instead is a function that can only be used by the owner.

```
function enableClaiming() external onlyOwner {
        require(lynoToken != address(0), "Token not set");
        claimingEnabled = true;
        emit ClaimingEnabled();
}
...
function claim() external nonReentrant {
        //...
        require(!hasClaimed[msg.sender], "Already claimed");
        //...
}
```

Instead the owner is able to receive rewards during purchase in case of ETH or by using the `withdrawERC20` function in case of tokens. This means that the owner can receive the contracts funds before users can ensure that they can receive their purchased tokens.

```
function buyWithETH() external payable nonReentrant
whenNotPaused {
        (bool sent, ) = payable(ethReceiver).call{value:
msg.value}("");
        require(sent, "ETH transfer failed");
}
...
function withdrawERC20(address token) external onlyOwner
```

Recommendation

The team should develop a more automated and less centralized approach for the claiming of tokens. A potential solution could be an introduction of a sofcap/hardcap for when users or owner are able to claim their tokens or funds respectively. Automated processes will ensure that users will not depend on the actions of the owner to claim their purchased tokens. Additionally, to enhance trust the owner should also be able to claim their funds only after the enabling of claiming for the users.

TSI - Tokens Sufficiency Insurance

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

Description

The `lynoTokens` need to be provided from an external source. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks. For example users are able to purchase tokens without being ensured that that they will receive them during claim.

```
function claim() external nonReentrant {
    //...
    IERC20(lynoToken).safeTransfer(msg.sender, amount);
    //...
}
```

Recommendation

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

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	LynoPresale.sol#L202,271,304,321,335
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
(, int256 price, , , ) = ethPriceFeed.latestRoundData();
IERC20(usdcAddress).safeTransferFrom(msg.sender, address(this),
amount);
IERC20(usdtAddress).safeTransferFrom(msg.sender, address(this),
amount);
IERC20(lynoToken).safeTransfer(msg.sender, amount);
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	LynoPresale.sol#L163
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.

```
address _lynoToken
```

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/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	LynoPresale.sol#L219,222,223,256,257,289,290
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 baseTokens = (normalizedAmount * PRECISION) /
stage.price
uint256 bonusTokens = (baseTokens * stage.bonusPercentage) /
100
```

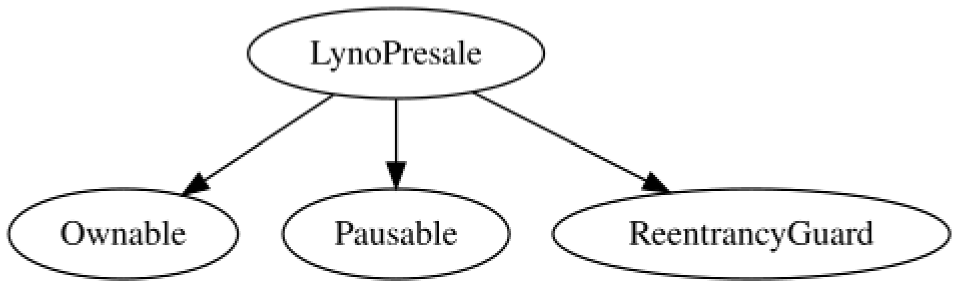
Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

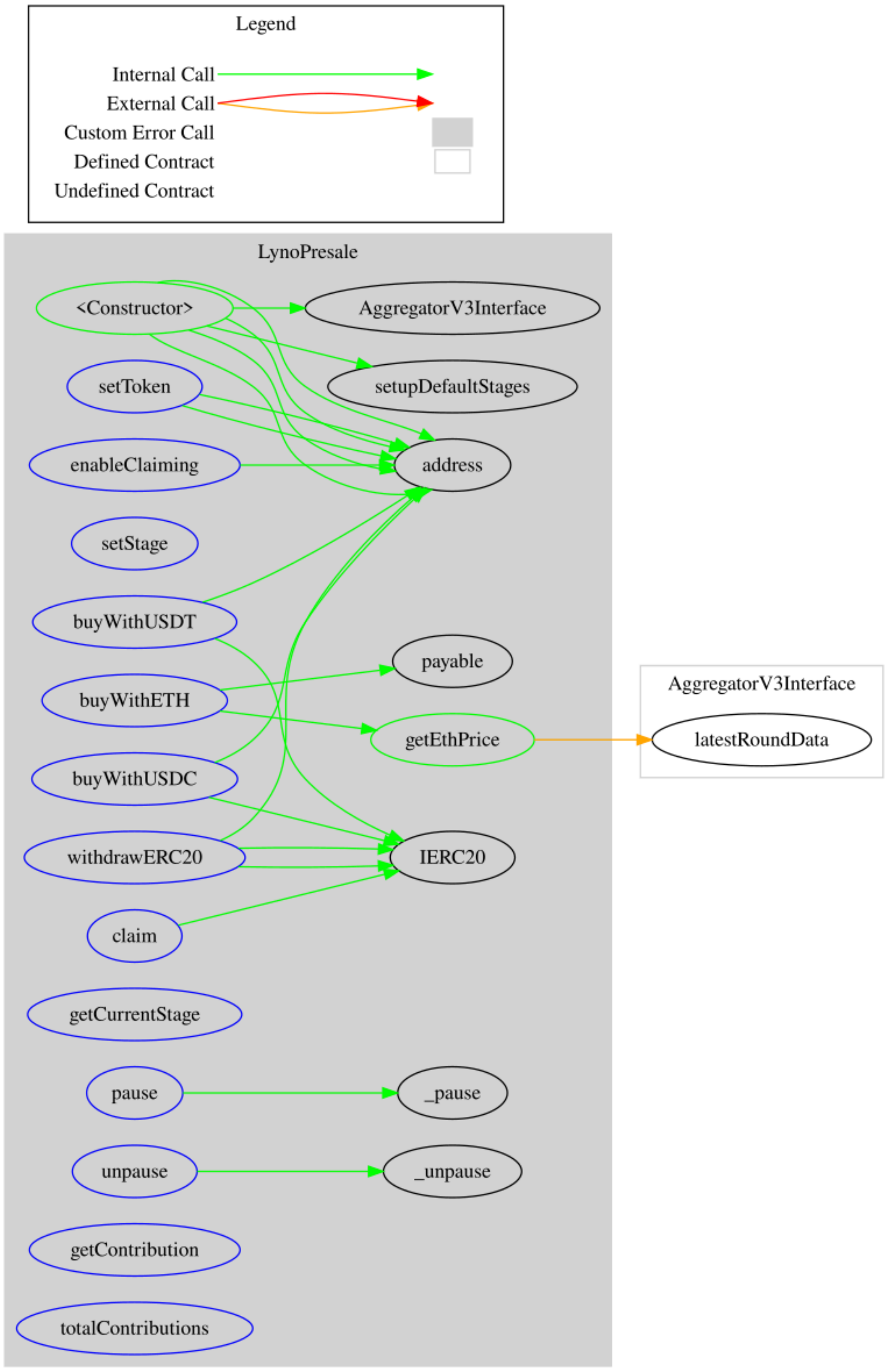
Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
LynoPresale	Implementation	Ownable, Pausable, ReentrancyGuard		
		Public	✓	Ownable
	setupDefaultStages	Private	✓	
	setToken	External	✓	onlyOwner
	enableClaiming	External	✓	onlyOwner
	setStage	External	✓	onlyOwner
	getEthPrice	Public		-
	buyWithETH	External	Payable	nonReentrant whenNotPaused
	buyWithUSDC	External	✓	nonReentrant whenNotPaused
	buyWithUSDT	External	✓	nonReentrant whenNotPaused
	claim	External	✓	nonReentrant
	withdrawERC20	External	✓	onlyOwner
	getCurrentStage	External		-
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	getContribution	External		-
	totalContributions	External		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
LynoPresale
<Constructor>
AggregatorV3Interface
setToken
setupDefaultStages
enableClaiming
address
setStage
buyWithUSDT
payable
buyWithETH
getEthPrice
buyWithUSDC
withdrawERC20
IERC20
claim
getCurrentStage
pause
_pause
unpause
_unpause
getContribution
totalContributions
AggregatorV3Interface
latestRoundData

Summary

Lyno AI contract implements a presale mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

About Cyberscope

Cyberscope is a TAC 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.

A **TAC Security** Company

The Cyberscope team

cyberscope.io