Cyberscope

Audit Report

Genesis Presale

June 2024

Repository https://github.com/0xGenesisDAO/genesisdao-contracts

Commit 5a85a693832b9dea70f1e60980b4536e0f0037a6

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/0xGenesisDAO/genesisdao-contracts
Commit	5a85a693832b9dea70f1e60980b4536e0f0037a6

Audit Updates

Initial Audit	19 Jun 2024
Corrected Phase 2	20 Jun 2024

Source Files

Filename	SHA256
Presale.sol	749d4d13eb5ed80e46c4f6375418a53c5bed4543217bb1650af1e4eaff1ebd2d
IPresale.sol	6f8d687b447026c0a627576d7cf7154886d6b416d41c7ba8f907c9dd6b990dac

Overview

The contract is designed to handle the presale of tokens, providing a structured process for users to participate by contributing native tokens (ETH) in exchange for the presale tokens. The contract ensures the presale's integrity by incorporating mechanisms for token deposits, user contributions, refunds, and the eventual launch of tokens on a decentralized exchange (Uniswap). The owner of the contract has specific administrative privileges to manage the presale effectively.

Token Purchase and Contributions

Users have the ability to participate in the presale by sending native tokens (ETH) to the contract. Contributions are tracked in a mapping, and the total amount of funds raised is recorded in the contract's state. This functionality is facilitated through the `receive()` function, which handles direct ETH transfers to the contract. Additionally, the `_purchase()` function validates each contribution, ensuring it adheres to the predefined minimum and maximum limits, and the presale's overall cap.

Claiming Tokens

After the presale is successfully finalized and launched, users can claim their allocated tokens. The `claim()` function allows contributors to retrieve their tokens based on their proportional contributions. This functionality ensures that users receive their due share of the presale tokens once the presale transitions to a claimable state.

Refund Mechanism

In scenarios where the presale is canceled or fails to reach the soft cap by the end date, users are entitled to refunds. The `refund()` function enables contributors to retrieve their contributions if the presale conditions are not met. This is protected by the `onlyRefundable` modifier, which ensures refunds are only processed under appropriate conditions, providing security to the participants.

Owner Privileges

The contract owner has significant control over the presale process. They can initialize the presale by depositing tokens through the `deposit()` function, finalize the presale with `finalize()`, and launch the presale on Uniswap using the `launch()` function. Additionally, the owner can cancel the presale with `cancel()`, extend the presale period with `extendEndTime()`, and unlock token claiming with `unlockClaim()`. These functions ensure the owner can manage the presale lifecycle effectively and respond to different scenarios.

Token Deposit and Allocation

The `deposit()` function, callable only by the owner, is used to deposit the total tokens intended for the presale and liquidity provision. This function sets the contract's state to active and calculates the distribution of tokens between liquidity and presale allocations. The `_tokensForLiquidity()` and `_tokensForPresale()` functions determine the precise token amounts for these purposes, ensuring a clear allocation strategy.

Finalization and Liquidity Provision

The `finalize()` function, also restricted to the owner, is used to conclude the presale if the soft cap is reached or the presale period ends. This function handles the allocation of tokens to seed investors, the calculation of liquidity provision, and the distribution of remaining ETH to the owner. The `_liquidityWei()` function calculates the ETH amount required for liquidity based on the presale and launch token prices.

Launch Process

To transition the presale tokens to the open market, the owner can call the `launch()` function. This function adds liquidity to Uniswap using the presale tokens and ETH, enabling the tokens to be traded publicly. The `_liquify()` function facilitates this process by interacting with the Uniswap router to create a liquidity pool, ensuring the tokens are properly listed and tradable.

Refunds and Token Claiming Post-Launch

After the presale is successfully launched, the owner can call `unlockClaim()` to enable contributors to claim their tokens. This function changes the state to allow claims, ensuring

that participants can retrieve their tokens. The `refund()` function remains available for scenarios where the presale is canceled or fails, providing a safety net for contributors.

Roles

Owner

The owner can interact with the following functions:

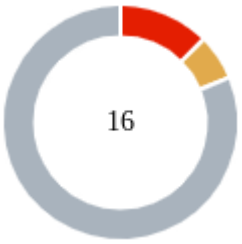
- function finalize
- function launch
- function cancel
- function extendEndTime
- function unlockClaim

Users

The users can interact with the following functions:

- function deposit
- function claim
- function refund

Findings Breakdown



Critical	2
Medium	1
Minor / Informative	13

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	1	0	0	0
Minor / Informative	13	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	ERWDF	Exploitable Refund Window During Finalization	Unresolved
	IRL	Improper Refund Logic	Unresolved
	PTU	Potential Token Underflow	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	EUI	Enum Usage Improvement	Unresolved
	ITE	Incorrect Time Extension	Unresolved
	MREM	Misleading Revert Error Message	Unresolved
	MEE	Missing Events Emission	Unresolved
	MPV	Missing Parameter Verification	Unresolved
	MU	Modifiers Usage	Unresolved
	PDTD	Potential Disproportionate Token Distribution	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	RC	Repetitive Calculations	Unresolved

●	L08	Tautology or Contradiction	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L19	Stable Compiler Version	Unresolved

ERWDF - Exploitable Refund Window During Finalization

Criticality	Critical
Location	Presale.sol#L79,131,178,
Status	Unresolved

Description

The contract allows the owner to call the `finalize` function even when the pool has ended but the funds have not been raised to the soft cap. Specifically, if `weiRaised < softCap` and `block.timestamp > pool.options.end`, the finalize function can still be executed because the `revert` condition in the if statement only checks if both conditions (funds not raised and pool time not ended) are true. This results in a situation where, after finalization, users can call the `refund` function due to the pool being in state 4 and the refund conditions being met (pool not reaching the soft cap and the pool time being ended). This can lead to users getting refunds while the contract attempts to add liquidity, which will fail due to insufficient funds, ultimately preventing the liquidity launch.

Exploit Scenario:

1. Pool Ends Without Reaching Soft Cap: The pool ends (block.timestamp > pool.options.end), but the funds raised (weiRaised) are less than the soft cap (softCap).
2. Owner Calls finalize: The owner calls the `finalize` function, changing the pool state to 4 and calculating the liquidity to be added, without actually adding the liquidity immediately.
3. Users Call refund: During the window between finalization and liquidity addition, users call the `refund` function. Since the pool state is 4 and the pool has not reached the soft cap, the refund conditions are met, allowing users to withdraw their funds.
4. Funds Are Depleted: As users withdraw their funds, the contract's funds are modified, making it impossible to add the calculated liquidity when the launch function is called.

5. Liquidity Launch Fails: When the owner tries to call the launch function, it fails because the required funds have been refunded to users, and the contract can no longer add the necessary liquidity.

```
    modifier onlyRefundable() {
        if (
            !(pool.state != 3 ||
                (block.timestamp > pool.options.end &&
                    pool.weiRaised < pool.options.softCap))
        ) revert NotRefundable();
        _;
    }

    function finalize() external onlyOwner returns(bool) {
        if(pool.state != 2) revert InvalidState(pool.state);
        if(pool.weiRaised < pool.options.softCap && block.timestamp <
pool.options.end) revert SoftCapNotReached();

        pool.state = 4;
        ...
        uint256 liquidityWei = _liquidityWei();
        ...
        }

    function launch() external onlyOwner {
        if (pool.state != 4) revert InvalidState(pool.state);
        pool.state = 5;

        uint256 liquidityWei = _liquidityWei();

        // Add LP
        _liquify(liquidityWei, pool.tokensLiquidity);
        pool.tokenBalance -= pool.tokensLiquidity;
    }
```

Recommendation

It is recommended to refactor the `finalize` function to include a check ensuring that the pool has reached its soft cap before allowing the finalization process. Additionally, the liquidity addition process should be made automatic, ensuring that once the pool is `finalized`, the liquidity is added immediately, preventing any window of time where users can exploit the `refund` function. This will ensure that the pool can be successfully

launched without the risk of funds being withdrawn by users post-finalization but pre-launch.

IRL - Improper Refund Logic

Criticality	Critical
Location	Presale.sol#L84
Status	Unresolved

Description

The contract is using the `onlyRefundable` modifier, which currently reverts the transaction if the pool state is not canceled, or if the current block timestamp is greater than the end time and the `weiRaised` amount is less than the `softCap`. This logic prevents users from receiving refunds in scenarios where the fundraising campaign has ended unsuccessfully, meaning the `softCap` was not met by the end time. This behavior contradicts the expected presale logic, where users should be eligible for refunds if the campaign fails to meet its softCap by the end time.

```
/// @notice Canceled or NOT softcapped and expired
modifier onlyRefundable() {
    if(pool.state != 3 || (block.timestamp > pool.options.end
&& pool.weiRaised < pool.options.softCap)) revert
NotRefundable();
    _;
}
```

Recommendation

It is recommended to reevaluate the usage of the `onlyRefundable` modifier. The contract should allow refunds if the `block timestamp` is greater than the `end` time and the `weiRaised` amount is less than the `softCap`, rather than reverting in such cases. This adjustment will ensure that the contract aligns with the intended business logic, providing users with refunds when a fundraising campaign does not meet its minimum funding requirements by the designated end time.

PTU - Potential Token Underflow

Criticality	Medium
Location	Presale.sol#L138,321,330
Status	Unresolved

Description

The contract is calculating `seederTokens`, which are the tokens owed to the seed investors, based on a percentage of the `tokensClaimable` value. These tokens are then deducted from the `pool.tokensLiquidity`. However, this deduction may result in an underflow if the `pool.tokensLiquidity` is not sufficient to cover the `seederTokens`, potentially leading to serious issues in the contract's operation and distribution of tokens. Specifically, if the `_seederTotalValue` is disproportionate to the available funds, the resulting `seederTokens` calculation could exceed the `pool.tokensLiquidity`. This mismatch may cause an underflow, which could disrupt the accurate allocation of tokens and adversely affect the contract's stability and functionality.

```
uint256 seederTokens = _seederTokens();
// Send seed investor tokens to owner, for further redistribution
IERC20(pool.token).safeTransfer(msg.sender, seederTokens);
pool.tokenBalance -= seederTokens;
// Subtract the seed investor tokens from the amount allocated to the LP
pool.tokensLiquidity -= seederTokens;
...
function _seederTokens() internal view returns(uint256) {
    return ((_seederTotalValue() * SCALE) / pool.weiRaised *
pool.tokensClaimable) / SCALE;
}

function _seederTotalValue() internal view returns (uint256) {
    return pool.options.seederRaisedWei + (pool.options.seederRaisedWei
* pool.options.seederPremiumBps / 10_000);
}
```

Recommendation

It is recommended to add additional checks to verify that `pool.tokensLiquidity` is greater than the `seederTokens` being subtracted. This will prevent underflow errors and ensure the contract operates securely and as intended, maintaining the integrity of token distributions.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Presale.sol#L90,110,131,163,179,198
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.

Specifically, the owner has significant control over the presale process. The owner has the authority to set the correct attributes and initialize the contract. Additionally, the owner can control the flow and process of the presale by calling functions such as `deposit`, `finalize`, `launch`, `cancel`, `extendEndTime`, and `unlockClaim` in correct flow. This centralization of control could potentially be exploited, leading to unfair practices or mismanagement.

```
   constructor (address _weth, address _token, address
_uniswapV2Router02, PresaleOptions memory _options)
Ownable(msg.sender) {
        _prevalidatePool(_options);
        pool.uniswapV2Router02 =
IUniswapV2Router02(_uniswapV2Router02);
        pool.routerAddress = _uniswapV2Router02;
        pool.token = IERC20(_token);
        pool.state = 1;
        pool.weth = _weth;
        pool.options = _options;
    }
    function deposit() external onlyOwner returns (uint256) {
        ...

        IERC20(pool.token).safeTransferFrom(msg.sender,
address(this), pool.options.tokenDeposit);

        emit Deposit(msg.sender, pool.options.tokenDeposit,
block.timestamp);
        return pool.options.tokenDeposit;
    }

    function finalize() external onlyOwner returns(bool) {
        ...
        if (withdrawable > 0)
payable(msg.sender).sendValue(withdrawable);

        emit Finalized(msg.sender, pool.weiRaised,
block.timestamp);

        return true;
    }


    function launch() external onlyOwner {
        ...
        _liquify(liquidityWei, pool.tokensLiquidity);
        pool.tokenBalance -= pool.tokensLiquidity;
    }


    function cancel() external onlyOwner returns(bool){
        ...
        emit Cancel(msg.sender, block.timestamp);

        return true;
    }

    function extendEndTime(uint112 end) external onlyOwner {
```

```
        ...
        pool.options.end = end;
    }
```

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.

EUI - Enum Usage Improvement

Criticality	Minor / Informative
Location	Presale.sol#L70
Status	Unresolved

Description

The contract is using a `uint8` variable to represent different states of the presale, which could lead to potential issues with readability and maintainability. Using an enumeration (`enum`) instead of a `uint8` for the state variable would make the code more understandable and reduce the risk of incorrect state assignments, as enums provide a more explicit representation of the different states.

```
    * @param state Current state of the presale {1: Initialized, 2:
Active, 3: Canceled, 4: Finalized, 5: Launched, 6: Claimable}.
     * @param options PresaleOptions struct containing configuration for
the presale.
     */
    struct Pool {
    ...
    uint8 state;

    ...
```

Recommendation

It is recommended to replace the `uint8` state variable with an enumeration (`enum`) to clearly define the different states of the presale. This will enhance code readability, improve maintainability, and minimize the risk of assigning invalid states.

ITE - Incorrect Time Extension

Criticality	Minor / Informative
Location	Presale.sol#L198
Status	Unresolved

Description

The contract is currently resetting the end time when extending the presale period using the `extendEndTime` function. Instead of adding additional time to the existing end time, the function replaces the current end time with a new value. This approach can be misleading and may not align with the expected behavior of extending the presale period, potentially causing confusion and mismanagement of the presale timeline.

```
    function extendEndTime(uint112 end) external onlyOwner {
        if(block.timestamp < pool.options.start ||
block.timestamp > pool.options.end) revert
NotInPurchasePeriod(); // Presale can only be extended during
active period
        if(pool.state != 2) revert InvalidState(pool.state);
        pool.options.end = end;
    }
```

Recommendation

It is recommended to modify the `extendEndTime` function to add the specified time to the existing `end` time, rather than resetting it. This will ensure that the presale period is accurately extended, preserving the intended duration and avoiding any potential issues related to time management.

MREM - Misleading Revert Error Message

Criticality	Minor / Informative
Location	Presale.sol#L133
Status	Unresolved

Description

The contract is using an error message "SoftCapNotReached" when checking if `pool.weiRaised < pool.options.softCap && block.timestamp < pool.options.end`. This check reverts the transaction if the `softCap` is not reached and the current time is before the end time. However, the same error message is also triggered if the `softCap` is reached but the current time is after the end time. This is misleading because it does not accurately reflect the actual issue, which could either be the `softCap` not being reached or the fundraising period having ended.

```
if(pool.weiRaised < pool.options.softCap && block.timestamp < pool.options.end) revert SoftCapNotReached();
```

Recommendation

It is recommended to revise the error handling logic to provide distinct and accurate error messages for different failure conditions. This will ensure that users and developers can clearly understand whether the issue is due to the `softCap` not being reached or the fundraising period having ended, thereby improving the transparency and usability of the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Presale.sol#L198,224
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 extendEndTime(uint112 end) external onlyOwner {
        if(block.timestamp < pool.options.start ||
block.timestamp > pool.options.end) revert
NotInPurchasePeriod(); // Presale can only be extended during
active period
        if(pool.state != 2) revert InvalidState(pool.state);
        pool.options.end = end;
    }

    function unlockClaim() external onlyOwner {
        if(pool.state != 5) revert InvalidState(pool.state);
        pool.state = 6;
    }
```

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.

MPV - Missing Parameter Verification

Criticality	Minor / Informative
Location	Presale.sol#L300
Status	Unresolved

Description

The contract is missing checks to verify parameters during the constructor initialization. Specifically, there are no checks to ensure that `seederPremiumBps` is not zero and less than 10,000, `seederRaisedWei` is not zero, and `launchIncreaseBps` is less than 10,000. These missing checks could lead to improper initialization of the contract and potential vulnerabilities in the contract's operation.

```
function _prevalidatePool(PresaleOptions memory _options)
internal view returns(bool) {
    if (_options.softCap == 0) revert InvalidCapValue();
    if (_options.min == 0 || _options.min > _options.max)
revert InvalidLimitValue();
    if (_options.liquidityBps < 0 || _options.liquidityBps >
10000) revert InvalidLiquidityValue();
    if (_options.start > block.timestamp || _options.end <
_options.start) revert InvalidTimestampValue(); // Comment this
line out for testing purpose
    return true;
}
```

Recommendation

It is recommended to include validation checks during the constructor initialization to ensure that `seederPremiumBps` is not zero and is less than 10,000, `seederRaisedWei` is not zero, and `launchIncreaseBps` is less than 10,000. Implementing these checks will enhance the security and robustness of the contract by preventing invalid parameter values from being used.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	Presale.sol#L132,200,288
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.

```
if(pool.state != 2) revert InvalidState(pool.state);
```

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.

PDTD - Potential Disproportionate Token Distribution

Criticality	Minor / Informative
Location	Presale.sol#L321,330
Status	Unresolved

Description

The contract calculates the `_seederTotalValue`, which represents the effective wei raised by seed investors. This value is then used to calculate `seederTokens`, the amount of tokens owed to seed investors. However, the contract should verify that `_seederTotalValue` is less than `pool.weiRaised`, otherwise, there would be no tokens left for other distributions. Additionally, if `_seederTotalValue` becomes disproportionate compared to `pool.weiRaised * pool.tokensClaimable`, the contract may not have enough tokens to send to seed investors, causing inacurate calcations and disrupting the accurate allocation and functionality of the contract.

```
    /**
     * @notice Tokens per seed investor rate is dynamically calculated
using the proportional allocation of current raise amount in Wei.
     * @return The total amount of tokens allocated to the seed
investors
     */
    function _seederTokens() internal view returns(uint256) {
        return ((_seederTotalValue() * SCALE) / pool.weiRaised *
pool.tokensClaimable) / SCALE;
    }

    /**
     * @notice Calculates the amount of effective wei the seeders have
raised.
     * For example, at a 30% premium (seederPremiumBps = 3000), and 10
ETH seed funds raised, the total value will be 13 ETH.
     * @return The amount of token value in wei the seeders have raised.
     */
    function _seederTotalValue() internal view returns (uint256) {
        return pool.options.seederRaisedWei +
(pool.options.seederRaisedWei * pool.options.seederPremiumBps / 10_000);
    }
```

Recommendation

It is recommended to include additional checks to ensure that `_seederTotalValue` is less than `pool.weiRaised` before proceeding with any token calculations. Moreover, ensure that the calculated `seederTokens` does not exceed the available tokens in the contract. These checks will prevent inacurate calculations and ensure the contract has sufficient tokens to fulfill its obligations, maintaining the integrity of token distribution.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Presale.sol#L118
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

Specifically the `pool.token` address may apply fee values to its transactions and as a result the presale contract will receive less tokens.

```
IERC20(pool.token).safeTransferFrom(msg.sender, address(this),
pool.options.tokenDeposit);
```

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
```

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	Presale.sol#L350,358
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

Speicifically, the contract is calculating tokens for liquidity using the `_tokensForLiquidity` function, which determines the amount based on a percentage of the total token deposit. However, within the `_tokensForPresale` function, these tokens for liquidity are recalculated instead of reusing the value derived from `_tokensForLiquidity`. This approach introduces a risk of inconsistency and potential errors in token allocation calculations.

```
   function _tokensForLiquidity() internal view returns
(uint256){
        return pool.options.tokenDeposit *
pool.options.liquidityBps / 10_000;
    }

    function _tokensForPresale() internal view returns
(uint256){
        return pool.options.tokenDeposit -
(pool.options.tokenDeposit * pool.options.liquidityBps /
10_000);
    }
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once

and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

It is recommended to reuse the `_tokensForLiquidity` function within the `_tokensForPresale` function to ensure consistency and accuracy in token allocation. This change will enhance the reliability of the contract by preventing potential discrepancies in the calculation of tokens for liquidity and presale.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	Presale.sol#L303
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
_options.liquidityBps < 0 || _options.liquidityBps > 10000
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Presale.sol#L314,322,340,341
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.

```
return ((contributions[contributor] * SCALE) / pool.weiRaised *
pool.tokensClaimable) / SCALE
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Presale.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

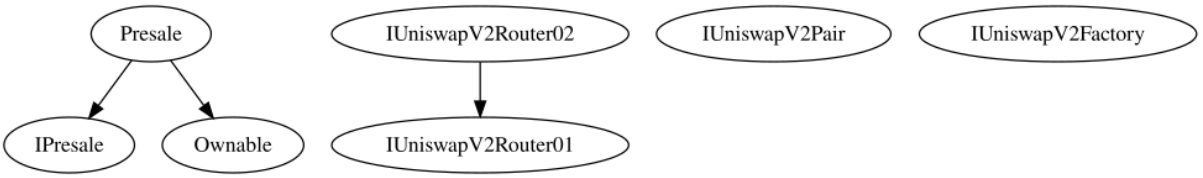
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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

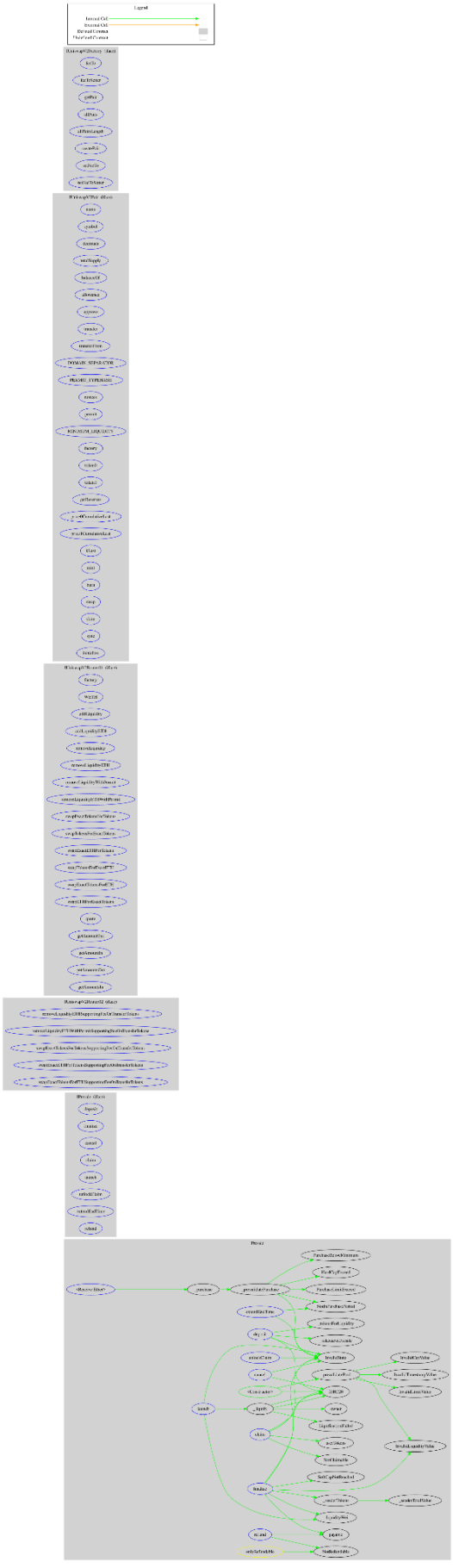
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Presale	Implementation	IPresale, Ownable		
		Public	✓	Ownable
		External	Payable	-
	deposit	External	✓	onlyOwner
	finalize	External	✓	onlyOwner
	launch	External	✓	onlyOwner
	cancel	External	✓	onlyOwner
	extendEndTime	External	✓	onlyOwner
	claim	External	✓	-
	unlockClaim	External	✓	onlyOwner
	refund	External	✓	onlyRefundable
	_purchase	Private	✓	
	_liquify	Private	✓	
	_prevalidatePurchase	Internal		
	_prevalidatePool	Internal		
	userTokens	Public		-
	_seederTokens	Internal		
	_seederTotalValue	Internal		
	_liquidityWei	Internal		

	_tokensForLiquidity	Internal		
	_tokensForPresale	Internal		
IPresale	Interface			
	deposit	External	✓	-
	finalize	External	✓	-
	cancel	External	✓	-
	claim	External	✓	-
	launch	External	✓	-
	unlockClaim	External	✓	-
	extendEndTime	External	✓	-
	refund	External	✓	-

Inheritance Graph



Flow Graph



Summary

The Genesis contract implements a comprehensive presale mechanism designed to facilitate token sales and liquidity provisioning in a secure and structured manner. This audit investigates potential security issues, examines the business logic for any concerns, and identifies possible improvements to enhance the overall functionality and reliability of the contract.

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