Cyberscope

Audit Report

Estiapayments

Nov 2024

Repository	https://github.com/Estiapayments/Estia/tree/audit-fix
Commit	fe3a6b74fb3c31f6926801e0bdb4c1a183dc18f3
Audited by	© cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **4**
Review **5**
Audit Updates 5
Source Files 5
Overview **6**
Estia Contract 6
EstiaCrowdSale Contract 6
Token Purchase Functionality 6
Vesting Mechanism 6
Crowdsale Timing 7
Finalization Functionality 7
Owner Functionalities 7
Withdrawal Functions 7
EstiaVesting Contract 8
Token Grant Functionality 8
Vesting and Claiming Mechanism 8
Revocation of Grants 8
Founders Vesting Rules 9
Grant Management and Update Functions 9
Withdraw Functions 9
Findings Breakdown **10**
Diagnostics **11**
IVL - Inadequate Vesting Lock 13
Description 13
Recommendation 13
CO - Code Optimization 14
Description 14
Recommendation 15
CCR - Contract Centralization Risk 17
Description 17
Recommendation 20
DPI - Decimals Precision Inconsistency 21
Description 21
Recommendation 21
FSNU - Founder Status Not Updated 23
Description 23
Recommendation 25
MMN - Misleading Method Naming 26

Description 26
Recommendation 26
MCU - Missing CrowdsaleRound Usage 27
Description 27
Recommendation 28
MTDV - Missing Token Decimal Verification 29
Description 29
Recommendation 29
MVIC - Missing Vesting Interface Check 30
Description 30
Recommendation 30
RFP - Redundant Founder Parameter 31
Description 31
Recommendation 33
RSML - Redundant SafeMath Library 34
Description 34
Recommendation 34
RTC - Redundant Time Check 35
Description 35
Recommendation 35
RTF - Redundant Timestamp Function 36
Description 36
Recommendation 36
RUC - Redundant uint256 Casting 37
Description 37
Recommendation 37
TSI - Tokens Sufficiency Insurance 38
Description 38
Recommendation 38
OCTD - Transfers Contract's Tokens 39
Description 39
Recommendation 39
ZAI - Zero Address Initialization 41
Description 41
Recommendation 41
L04 - Conformance to Solidity Naming Conventions 43
Description 43
Recommendation 44
L06 - Missing Events Access Control 45
Description 45
Recommendation 45
L09 - Dead Code Elimination 46

Description 46
Recommendation 46
L13 - Divide before Multiply Operation 47
Description 47
Recommendation 47
L15 - Local Scope Variable Shadowing 48
Description 48
Recommendation 48
L20 - Succeeded Transfer Check 49
Description 49
Recommendation 49
Functions Analysis **50**
Inheritance Graph **55**
Flow Graph **56**
Summary **57**
Disclaimer **58**
About Cyberscope **59**

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/Estiapayments/Estia/tree/audit-fix
Commit	fe3a6b74fb3c31f6926801e0bdb4c1a183dc18f3

Audit Updates

Initial Audit	20 Sep 2024 https://github.com/cyberscope-io/audits/blob/main/4-est/v1/audit.pdf
Corrected Phase 2	15 Oct 2024 https://github.com/cyberscope-io/audits/blob/main/4-est/v2/audit.pdf
Corrected Phase 3	13 Nov 2024

Source Files

Filename	SHA256
EstiaVesting.sol	05c72eb535d33f3941958c83178501b9890426f46182b18b024527a7d896afc4
EstiaCrowdSale.sol	d4c29cb3b2483d55eb8add45c0092fd323531ece3c074a8f78e4b00f708c0062
Estia.sol	273a2a9db706b3dd7a592ce594f02bf9fa67f4227c1c0a76f65565d927f0ff79

Overview

Estia Contract

The Estia contract implements a decentralized token with upgradeable functionalities, leveraging OpenZeppelin's upgradeable contracts. It is an ERC20 token named "Estia" (symbol: EST), which includes features such as ownership control, permit-based approvals (ERC20Permit), and the ability to be upgraded via the UUPS upgradeable pattern. The contract includes a special function, `airdrop`, allowing the owner to distribute tokens to multiple recipients in a single transaction. It emits an `Airdrop` event after successfully transferring tokens to specified addresses. The contract is initialized with a total supply of 260 million EST tokens.

EstiaCrowdSale Contract

The EstiaCrowdSale contract implements a comprehensive and secure crowdsale mechanism for distributing the Estia tokens (EST) in exchange for USDT contributions. It integrates features such as rate setting, time-based sales, finalization, and vesting of purchased tokens. This contract ensures secure token purchases, flexible sale configurations, and provides the owner with control over the crowdsale parameters.

Token Purchase Functionality

The core functionality of the contract is the `buyToken` function, which allows users to purchase Estia tokens in exchange for USDT. It validates the purchase, transfers the USDT from the user to the contract, and calculates the number of Estia tokens based on the current rate. The purchased tokens are then granted to the user with a vesting schedule. This process is logged through the `TokenPurchase` event, which records the purchaser, beneficiary, the USDT amount, and the number of tokens bought.

Vesting Mechanism

Purchased Estia tokens are subject to a vesting schedule, which ensures that tokens are released gradually over a set period. The vesting process is initialized with parameters such as vesting duration and an initial lock-in period, providing a structured token release

mechanism. This is handled through the `vestingToken.addTokenGrant` function, which sets up the vesting parameters for each purchase.

Crowdsale Timing

The `TimedCrowdsale` functionality enforces a specific timeframe within which the crowdsale is active. The contract sets `openingTime` and `closingTime`, and only allows token purchases within this window. If the sale period is extended, the `TimedCrowdsaleExtended` event is emitted, providing flexibility in adjusting the sale duration based on market conditions.

Finalization Functionality

Upon conclusion of the crowdsale, the `finalization` function allows the owner to perform additional operations, such as transferring remaining tokens back to the owner. The contract emits a `Finalized` event, indicating the completion of the crowdsale. This function ensures that no tokens are left stranded in the contract and marks the official end of the crowdsale.

Owner Functionalities

The owner has control over several key aspects of the contract, including pausing/unpausing the contract, extending the sale duration, and adjusting the rate, vesting period, and initial lock-in period. The owner can also update the reward token, USDT token, and withdraw any remaining tokens or Ether from the contract. These functions ensure that the owner can adapt the crowdsale parameters as necessary while maintaining security and operational integrity.

Withdrawal Functions

The contract includes secure mechanisms for withdrawing both ERC-20 tokens and Ether. The owner can withdraw any ERC-20 tokens held by the contract through the `withdrawToken` function, or withdraw Ether using the `withdrawEther` function. These features allow the owner to manage funds effectively after the crowdsale or in case of refunds.

EstiaVesting Contract

The EstiaVesting contract provides a secure and flexible mechanism for managing the vesting of Estia tokens (EST) to various recipients. It supports the creation, management, and revocation of token grants with customizable vesting schedules. The contract ensures that tokens are distributed over time according to predefined conditions, offering controlled and transparent token release to recipients, such as investors, founders, and other stakeholders.

Token Grant Functionality

The core functionality of the EstiaVesting contract revolves around the `addTokenGrant` function, which allows the `EstiaCrowdSale` contract to add or update token grants for recipients. Each grant is defined by parameters such as the recipient address, total token amount, initial lock period, vesting duration, the specific crowdsale round and the a boolean value indicating if the beneficiary is a founder. This function ensures that tokens are gradually released over the vesting period, preventing recipients from claiming all tokens upfront. If a recipient already has an existing grant, the new tokens are added to their grant, and the `GrantUpdateAmount` event is emitted to reflect this update.

Vesting and Claiming Mechanism

Recipients can claim their vested tokens using the `claimVestedTokens` function. This function calculates the number of tokens that have vested based on the elapsed time and updates the recipient's grant record accordingly. If the claimed amount is greater than zero, the contract transfers the tokens to the recipient and emits the `GrantTokensClaimed` event. This functionality ensures that recipients receive tokens only after they are fully vested, providing a controlled release of the token supply.

Revocation of Grants

The owner of the contract have the ability to revoke token grants using the `revokeTokenGrant` function. This functionality is useful if the grant needs to be canceled due to changes in circumstances. When a grant is revoked, the contract calculates the amount of tokens that have vested and returns the non-vested tokens to the `crowdsale_address` address. The `GrantRevoked` event is emitted to log the

details of the revoked grant, including the recipient address and the amounts of vested and non-vested tokens.

Founders Vesting Rules

The contract includes special provisions for founders, with specific vesting and lock-in periods defined through the `addFounders` function. This function is called by the owner and allows for the allocation of tokens to exactly six founder addresses, each time the function is called. Each founder is assigned an equal portion of the total token amount, along with a unique vesting duration and lock-in period. Founders have their grants marked by the `isFounder` flag, differentiating them from other recipients. The owner can update the amount and vesting duration for founders if a grant already exists, or create a new grant if necessary. Founders' token grants are initiated with a specified lock-in period before vesting begins, controlled by the `foundersLockInPeriodInSeconds`. This mechanism ensures that founders receive a tailored vesting schedule, distinct from regular recipients, and the vesting details are adjusted accordingly by the owner through the `addFounders` function.

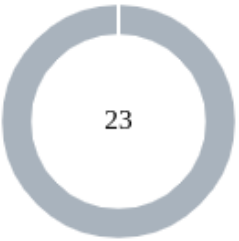
Grant Management and Update Functions

The contract owner has several functions to manage and update grants. The `changeStartTime` function allows updating the vesting start time for multiple recipients in a specific crowdsale round. The `updateIntervalTime` function can modify the interval time between token distributions, ensuring flexibility in how and when tokens are released. These functions provide the owner with control over the vesting schedule and distribution intervals, enabling adjustments to accommodate changing requirements.

Withdraw Functions

The contract includes secure withdrawal functions for both ERC-20 tokens and Ether. The `withdrawToken` function allows the owner to withdraw any ERC-20 tokens from the contract, while the `withdrawEther` function enables the withdrawal of Ether. These functions ensure that the owner can recover funds or tokens from the contract if necessary, adding an additional layer of control over the contract’s assets.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	23

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	23	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	IVL	Inadequate Vesting Lock	Unresolved
	CO	Code Optimization	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	DPI	Decimals Precision Inconsistency	Unresolved
	FSNU	Founder Status Not Updated	Unresolved
	MMN	Misleading Method Naming	Unresolved
	MCU	Missing CrowdsaleRound Usage	Unresolved
	MTDV	Missing Token Decimal Verification	Unresolved
	MVIC	Missing Vesting Interface Check	Unresolved
	RFP	Redundant Founder Parameter	Unresolved
	RSML	Redundant SafeMath Library	Unresolved
	RTC	Redundant Time Check	Unresolved
	RTF	Redundant Timestamp Function	Unresolved
	RUC	Redundant uint256 Casting	Unresolved

●	TSI	Tokens Sufficiency Insurance	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	ZAI	Zero Address Initialization	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

IVL - Inadequate Vesting Lock

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L356
Status	Unresolved

Description

The contract is setting a vesting lock timeframe during the presale functionality, but this lock only applies for a specific period. As a result, while the presale is still ongoing, users may be able to claim their tokens prematurely, which undermines the purpose of a vesting schedule. This behavior could lead to users accessing their tokens before the presale functionality ends, creating an imbalance in the distribution process and potential risks for other investors.

```
    function initialize(
        uint256 rate,
        IERC20 _token,
        IERC20 _usdtToken,
        uint256 openingTime,
        uint256 closingTime,
        EstiaVesting vesting // the vesting contract
    ) public initializer {
        round = 1;
        vestingToken = vesting;
        vestingMonths = 2;
        initialLockInPeriodInSeconds = 300 seconds;
        ...
```

Recommendation

It is recommended to implement a check that prevents any token claims before the presale has concluded. This would ensure that the vesting lock is effective, and users cannot claim tokens until after the presale is finished, maintaining the integrity of the vesting process and protecting the interests of all participants.

CO - Code Optimization

Criticality	Minor / Informative
Location	EstiaVesting.sol#L475
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

Specifically, in the `calculateGrantClaim` function, the `amountVestedPerMonth` is recalculated each time the function is called. This is done by subtracting the `totalClaimed` tokens and `monthsClaimed` months from the total grant amount and vesting duration, respectively. While this calculation yields the correct result, it introduces unnecessary computational overhead since the `amountVestedPerMonth` value can be derived once when the grant is initialized and stored for future use.

The calculation `tokenGrant.amount.sub(tokenGrant.totalClaimed).div(tokenGrant.vestingDuration.sub(tokenGrant.monthsClaimed))` is inefficient because it is recalculated every time the function is invoked, even though this value is consistent across calls, assuming a linear vesting schedule.

```
  function calculateGrantClaim(
        address _recipient,
        uint256 crowdsaleRound
    ) public view returns (uint256, uint256) {
        ...
        uint256 elapsedMonths = currentTime()
            .sub(tokenGrant.startTime)
            .div(monthTimeInSeconds)
            .add(1);

        // If over vesting duration, all tokens vested
        if (elapsedMonths > tokenGrant.vestingDuration) {
            uint256 remainingGrant = tokenGrant.amount.sub(
                tokenGrant.totalClaimed
            );
            uint256 balanceMonth = tokenGrant.vestingDuration.sub(
                tokenGrant.monthsClaimed
            );
            return (balanceMonth, remainingGrant);
        }

        // Calculate months vested and amount vested
        uint256 monthsVested =
elapsedMonths.sub(tokenGrant.monthsClaimed);
        uint256 amountVestedPerMonth = tokenGrant
            .amount
            .sub(tokenGrant.totalClaimed)

.div(tokenGrant.vestingDuration.sub(tokenGrant.monthsClaimed));
        uint256 amountVested = monthsVested.mul(amountVestedPerMonth);

        return (monthsVested, amountVested);
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

It is recommended to compute the `amountVestedPerMonth` as `tokenGrant.amount / tokenGrant.vestingDuration` when the grant is created and store it in the `tokenGrant` struct. By storing this precomputed value, the contract can avoid recalculating it in every call to `calculateGrantClaim`. This optimization will

not only reduce the computational load but also simplify the function’s logic, making it more gas-efficient and improving overall performance.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Estia.sol#L51 EstiaCrowdSale.sol#L434,459,473 EstiaVesting.sol#L327,345,426,624,670
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 can unilaterally change critical parameters such as the tokens used, the exchange rate, and the vesting schedules. Additionally, they have the ability to control token distribution through airdrops and designate or update the status of founders. This level of control creates a single point of failure and exposes the contract to potential misuse or decisions that may not align with the best interests of the community or token holders.

Specifically, the owner can unilaterally change critical parameters such as the tokens used, the exchange rate, and the vesting schedules. Additionally, they have the ability to control token distribution through airdrops, designate or update the status of founders, and revoke token grants. The owner has the authority to terminate a token grant, transferring all vested tokens to the `_recipient` while reclaiming any unvested tokens. Furthermore, the owner can update the `crowdsaleAddress` and then add the or update a token grant for a recipient in a specific crowdsale round. This level of control creates a single point of failure and exposes the contract to potential misuse or decisions that may not align with the best interests of the community or token holders.

```
function airdrop(
      address[] calldata recipients,
      uint256[] calldata amounts
  ) external onlyOwner {
      ...
      for (uint256 i = 0; i < recipients.length; i++) {
          transfer(recipients[i], amounts[i]);
      }
      emit Airdrop(msg.sender, recipients, amounts);
  }
```

```
function changeRate(
      uint256 newRate
  ) external virtual onlyOwner onlyWhileOpen whenNotPaused {
      require(newRate > 0, "Rate: Amount cannot be 0");
      _changeRate(newRate);
  }
  function changeToken(
      IERC20 newToken
  ) external virtual onlyOwner onlyWhileOpen whenNotPaused {
      require(
          address(newToken) != address(0),
          "Token: Address cant be zero address"
      );
      _changeToken(newToken);
  }
  function changeUsdtToken(
      IERC20 _usdtToken
  ) external virtual onlyOwner onlyWhileOpen whenNotPaused {
      _changeUsdtToken(_usdtToken);
  }
```

```
   function addCrowdsaleAddress(address crowdsaleAddress) external
onlyOwner {
        require(
            crowdsaleAddress != address(0),
            "ERC20: transfer from the zero address"
        );
        crowdsale_address = crowdsaleAddress;
    }

    function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this initial holding
period(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 crowdsaleRound, //1
        bool _isFounder
    ) external nonReentrant onlyCrowdsale {
    ...
    }

    function revokeTokenGrant(
        address _recipient,
        uint256 crowdsaleRound
    ) external nonReentrant onlyOwner{
        Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
        uint256 monthsVested;
        uint256 amountVested;
        (monthsVested, amountVested) = calculateGrantClaim(
            _recipient,
            crowdsaleRound
        );
        ...
        }

 function changeStartTime(
        address[] calldata _recipient,
        uint256[] calldata _startTime,
        uint256[] calldata _totalVestingMonths,
        uint256 crowdsaleRound
    ) external onlyOwner nonReentrant {
        require(_recipient.length == _startTime.length, "Invalid
parameters");
        require(_recipient.length == _totalVestingMonths.length,
"Invalid parameters");
        ...

            tokenGrant.startTime = _startTime[index];
            tokenGrant.vestingDuration = _totalVestingMonths[index];
```

```
        }
        emit StartTimeChanged(_recipient, _startTime,
_totalVestingMonths);
    }

 function addFounders(
        address[] calldata _founders,
        uint256 _amount,
        uint256 _vestingDurationInMonths,
        uint256 _round
    ) external onlyOwner nonReentrant {
        ...
            tokenGrants[_founders[i]][_round] = grant;
        }

        emit AddFounders(
            _founders,
            _amount,
            _vestingDurationInMonths,
            foundersLockInPeriodInSeconds,
            _round
        );
    }
```

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	EstiaCrowdSale.sol#L165
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 _getTokenAmount(
        uint256 _usdtAmount
    ) internal view returns (uint256) {
       uint256 tRate = (rate * 10 ** 6) / 10000; // Scale the rate for 6
decimals; rate is provided with up to 2 decimal places (hence division
by 10000).
        uint256 tokens = (_usdtAmount * tRate) / 10 ** 6;
        return tokens * 10 ** 12; // here 10**12 usdt is 6 decimal while
convert to Estia need to add 12 decimal.
    }
```

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.

FSNU - Founder Status Not Updated

Criticality	Minor / Informative
Location	EstiaVesting.sol#L345,659
Status	Unresolved

Description

The contract is designed to allow the addition of users as founders through the `addFounders` function. However, if a user already has an existing token grant and is later added as a founder, the contract does not update their `isFounder` status. The `addFounders` function only updates the amount and vesting duration of the existing grant, but it does not modify the `isFounder` flag to reflect the user's new status as a founder. As a result, even though the `addFounders` function may be executed for an existing grant holder, their `isFounder` value will remain false, leading to potential inconsistencies in user roles and permissions.

```
   function addTokenGrant(
        ...
        bool _isFounder
    ) external nonReentrant onlyCrowdsale {
        ...

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            Grant memory grant = Grant({
                ...
                isFounder: _isFounder
            });
            tokenGrants[_recipient][crowdsaleRound] = grant;
            emit GrantAdded(_recipient);
        }
    ...
    }

  function addFounders(
        address[] calldata _founders,
        uint256 _amount,
        uint256 _vestingDurationInMonths,
        uint256 _round
    ) external onlyOwner nonReentrant {
        require(_founders.length == 6, "Allowed only 6 founders");
        founders = _founders;

        uint256 amountPerFounder = _amount / 6; // Divide the total
amount equally among founders

        for (uint256 i = 0; i < _founders.length; i++) {
            Grant storage grant = tokenGrants[_founders[i]][_round]; //
Use 'storage' to modify the grant in-place

            // If the grant already exists, add the new amount and
update the vesting duration.
            if (grant.recipient == _founders[i]) {
                grant.amount += amountPerFounder; // Add to existing
amount
                grant.vestingDuration = _vestingDurationInMonths; //
Update vesting duration
            }
            ....
            }
```

Recommendation

It is recommended to ensure that when a user is added as a founder, their `isFounder` status is updated, regardless of whether they already hold a token grant. This can be achieved by modifying the `addFounders` function to explicitly update the `isFounder` flag within the user's grant. Doing so will ensure accurate role assignments and prevent misrepresentation of the founder status for existing grant holders.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L423
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

Specifically, the `finalization` function, despite its name, withdraws the contract's token balance to the owner instead of performing any finalization process.

```
    function finalization() internal virtual override {
        uint256 balance = rewardToken.balanceOf(address(this));
        require(balance > 0, "Finalization: Insufficient token
balance");
        rewardToken.transfer(owner(), balance);
    }/
```

Recommendation

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

MCU - Missing CrowdsaleRound Usage

Criticality	Minor / Informative
Location	EstiaVesting.sol#L345
Status	Unresolved

Description

The contract is utilizing the `crowdsaleRound` variable, which is intended to distinguish between different rounds of the crowdsale. However, there is no practical usage or differentiation based on this variable within the contract's logic. All rounds effectively perform the same function without any distinct behavior or conditions. This lack of functionality undermines the purpose of having separate crowdsale rounds and can lead to confusion or misinterpretation of the contract's intended behavior. It also increases the complexity of the contract without providing any additional value.

```
  function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this initial holding
period(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 crowdsaleRound, //1
        bool _isFounder
    ) external nonReentrant onlyCrowdsale {
        require(_recipient != address(0), "Invalid recipient address");
        require(
            !tokenGrants[_recipient][crowdsaleRound].isFounder &&
!_isFounder,
            "Founder can't able to participate."
        );
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_vestingDurationInMonths != 0, "Vesting duration cannot
be 0");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        ...
    }
```

Recommendation

It is recommended to reconsider the utilization of the `crowdsaleRound` variable. If the contract is meant to handle all rounds in the same manner, and there are no distinct differences between them, consider removing this variable to simplify the code. Otherwise, the contract should implement specific logic or conditions that differentiate each round based on the `crowdsaleRound` value. This could include different token distribution rules, varying vesting schedules, or any other conditions that provide a meaningful distinction between the rounds.

MTDV - Missing Token Decimal Verification

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L73,203
Status	Unresolved

Description

The contract is missing a consistency check for the token's decimal places during initialization. While the `_changeToken` function verifies that the new token has 18 decimals before making a change, there is no equivalent check during the initial setup in the `__Crowdsale_init_unchained` function. This oversight could lead to scenarios where the initial token lacks the expected 18 decimals, potentially causing incorrect calculations or issues within the contract's reward and distribution mechanisms if a token with different decimal precision is used.

```
   function __Crowdsale_init_unchained(uint256 _rate, IERC20 _token)
internal {
        require(_rate > 0, "Rate cant be 0");

        rate = _rate;
        rewardToken = _token;
    }
...
    function _changeToken(IERC20Extented newToken) internal virtual {
        require(newToken.decimals() == 18, "Token must have 18
decimals");
        rewardToken = newToken;
    }
```

Recommendation

It is recommended to include a check that verifies the token has 18 decimals during the initialization phase as well. Ensuring this consistency from the start will help prevent discrepancies and ensure uniform behavior across all functions relying on the token's decimal precision.

MVIC - Missing Vesting Interface Check

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L459
Status	Unresolved

Description

The contract is missing a critical check to ensure that the `newToken` provided in the `changeToken` function complies with the vesting interface. Without this verification, a non-vesting token could be set, which could cause issues with the vesting logic and potentially result in users not receiving their tokens as expected under the vesting schedule.

```
    function changeToken(
         IERC20 newToken
     ) external virtual onlyOwner onlyWhileOpen whenNotPaused {
         require(
             address(newToken) != address(0),
             "Token: Address cant be zero address"
         );
         _changeToken(newToken);
     }
```

Recommendation

It is recommended to implement a check within the `changeToken` function to ensure that the `newToken` adheres to the vesting interface. This will safeguard the integrity of the vesting process and prevent the contract from accepting tokens that do not support vesting, ensuring users receive their tokens according to the established schedule.

RFP - Redundant Founder Parameter

Criticality	Minor / Informative
Location	EstiaVesting.sol#L345 EstiaCrowdSale.sol#L108
Status	Unresolved

Description

The contract is using the `addTokenGrant` function to assign token grants, with a parameter `_isFounder` that is always passed as `false` when called by the `EstiaCrowdSale` contract. Since this function is exclusively called by `EstiaCrowdSale` and the `_isFounder` value is consistently set to `false`, the parameter is redundant. The contract could streamline the logic by internally setting the `isFounder` value to `false` instead of accepting it as a parameter. This would simplify the function's interface and reduce the risk of unnecessary parameter mismanagement.

```
        vestingToken.addTokenGrant(
            _beneficiary,
            tokens,
            initialLockInPeriodInSeconds,
            vestingMonths,
            round,
            false
        );
        ...

    function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this initial holding
period(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 crowdsaleRound, //1
        bool _isFounder
    ) external nonReentrant onlyCrowdsale {
        require(_recipient != address(0), "Invalid recipient address");
        require(
            !tokenGrants[_recipient][crowdsaleRound].isFounder &&
!_isFounder,
            "Founder can't able to participate."
        );
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_vestingDurationInMonths != 0, "Vesting duration cannot
be 0");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            Grant memory grant = Grant({
                startTime:
currentTime().add(initialLock).add(intervalTime),
                amount: _amount,
                vestingDuration: _vestingDurationInMonths,
                monthsClaimed: 0,
                totalClaimed: 0,
                recipient: _recipient,
                isFounder: _isFounder
            });
        ...
        }
```

Recommendation

It is recommended to remove the `_isFounder` parameter from the `addTokenGrant` function and hardcode its value to `false` within the function. This change will improve code clarity, reduce complexity, and eliminate the need for redundant parameter passing, ensuring the function is leaner and less prone to errors.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	EstiaVesting.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RTC - Redundant Time Check

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L282
Status	Unresolved

Description

The contract is utilizing two `require` statements to validate the `newClosingTime` parameter when extending the closing time of the crowdsale. The first check ensures that `newClosingTime` is greater than or equal to the `openingTime`, while the second check verifies that it is greater than the current `closingTime`. However, since the `closingTime` has already been validated to be greater than the `openingTime` when it was set, the first check becomes redundant. This redundancy adds unnecessary gas costs and increases the complexity of the function without providing additional security or logic enhancement.

```
    function _extendTime(uint256 newClosingTime) internal {
        require(
            newClosingTime >= openingTime,
            "Closing time cant be before opening time"
        );
        require(
            newClosingTime > closingTime,
            "New closing time must be greater than current closing time"
        );
        closingTime = newClosingTime;
        emit TimedCrowdsaleExtended(closingTime, newClosingTime);
    }
```

Recommendation

It is recommended to remove the first `require` statement that checks if the `newClosingTime` is greater than or equal to the `openingTime`, as this validation is redundant once the contract has already ensured that the `closingTime` is properly set during initialization. This will streamline the function, reduce gas costs, and improve the contract's efficiency without compromising its logic.

RTF - Redundant Timestamp Function

Criticality	Minor / Informative
Location	EstiaVesting.sol#L487
Status	Unresolved

Description

The contract contains a `currentTime` function that simply returns the `block.timestamp`. This function is redundant as the `block.timestamp` can be accessed directly in Solidity, and there is no additional logic or modification applied by this function.

```
if (currentTime() < tokenGrant.startTime) {
    return (0, 0);
}
```

Recommendation

It is recommended to remove the `currentTime` function and directly use `block.timestamp` wherever needed. This will simplify the code and avoid unnecessary function calls, improving readability and efficiency.

RUC - Redundant uint256 Casting

Criticality	Minor / Informative
Location	EstiaVesting.sol#L387,403
Status	Unresolved

Description

The contract contains redundant `uint256` casting for variables that are already of type `uint256`. The use of `uint256` casting in lines such as `tokenGrant.amount = uint256(tokenGrant.amount.add(_amount))` is unnecessary, as these variables are already defined as `uint256`. This adds extra complexity and decreases code readability without any functional benefit.

```
tokenGrant.amount = uint256(tokenGrant.amount.add(_amount));
...
tokenGrant.monthsClaimed = uint256(
    tokenGrant.monthsClaimed.add(monthsVested)
);
tokenGrant.totalClaimed = uint256(
    tokenGrant.totalClaimed.add(amountVested)
);
```

Recommendation

It is recommended to remove the redundant `uint256` casting from the affected variables. This will simplify the code, improve readability, and maintain clean coding practices without changing the functionality of the contract.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L141
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
function _deliverTokens(
    address _beneficiary,
    uint256 _tokenAmount
) internal {
    rewardToken.transfer(_beneficiary, _tokenAmount);
}
```

Recommendation

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

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L581
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `withdrawToken` function.

```
    function withdrawToken(
        address _tokenContract,
        uint256 _amount
    ) external onlyOwner nonReentrant {
        require(_tokenContract != address(0), "Address cant be zero
address");
        IERC20 tokenContract = IERC20(_tokenContract);
        tokenContract.safeTransfer(msg.sender, _amount);
        emit WithdrawToken(_tokenContract, msg.sender, _amount);
    }
```

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.

ZAI - Zero Address Initialization

Criticality	Minor / Informative
Location	EstiaVesting.sol#L300,327
Status	Unresolved

Description

The contract is using the `crowdsale_address` variable, which is initially set to the zero address and can only be updated through the `addCrowdsaleAddress` function. Until this function is called, the `crowdsale_address` remains set to the zero address. This poses a risk as any operations involving the `crowdsale_address` before it is updated would default to using the zero address. For example, if tokens are transferred to this (zero) address, they will be irreversibly lost. Furthermore, the `addCrowdsaleAddress` function itself includes a check that prevents setting the `crowdsale_address` to the zero address, creating a logical inconsistency.

```
    address public crowdsale_address;
    ...

    function addCrowdsaleAddress(address crowdsaleAddress) external
onlyOwner {
        require(
            crowdsaleAddress != address(0),
            "ERC20: transfer from the zero address"
        );
        crowdsale_address = crowdsaleAddress;
    }
```

Recommendation

It is recommended to initialize the `crowdsale_address` to a meaningful non-zero address, or to enforce additional checks and logic that prevent any operations from occurring while it is set to the zero address. Alternatively, consider requiring the `addCrowdsaleAddress` function to be called during contract deployment or initialization to ensure the `crowdsale_address` is correctly set before any related operations are

executed. This will prevent potential loss of tokens and ensure the contract operates as intended.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	EstiaVesting.sol#L300,308,346,347,349,351,415,427,457,465,476,524,541,553,582,583,597,598,625,626,627,671,672,673,674 EstiaCrowdSale.sol#L73,93,251,252,253,358,359,404,474,486,487,501,502
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 public crowdsale_address
IERC20 _token
address _recipient
uint256 _amount
uint256 _vestingDurationInMonths
bool _isFounder
uint256 _intervalTime
address _tokenContract
address payable _to
address[] calldata _recipient
uint256[] calldata _startTime
uint256[] calldata _totalVestingMonths
address[] calldata _founders
uint256 _round

...
```

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.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	EstiaVesting.sol#L332
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
crowdsale_address = crowdsaleAddress
```

Recommendation

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

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

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L329,331
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function finalization() internal virtual {}

function _updateFinalization() internal {
        isFinalized = false;
    }
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	EstiaVesting.sol#L510,514 EstiaCrowdSale.sol#L168,169,170
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 amountVestedPerMonth = tokenGrant
            .amount
            .sub(tokenGrant.totalClaimed)

.div(tokenGrant.vestingDuration.sub(tokenGrant.monthsClaimed))
uint256 amountVested = monthsVested.mul(amountVestedPerMonth)

uint256 tRate = (rate * 10 ** 6) / 10000
uint256 tokens = (_usdtAmount * tRate) / 10 ** 6
```

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.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L357,360,361
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
uint256 rate
uint256 openingTime
uint256 closingTime
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	EstiaVesting.sol#L411,449,452 EstiaCrowdSale.sol#L145,416
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(tokenGrant.recipient, amountVested)
token.transfer(crowdsale_address, amountNotVested)
token.transfer(_recipient, amountVested)
rewardToken.transfer(_beneficiary, _tokenAmount)
rewardToken.transfer(owner(), balance)
```

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

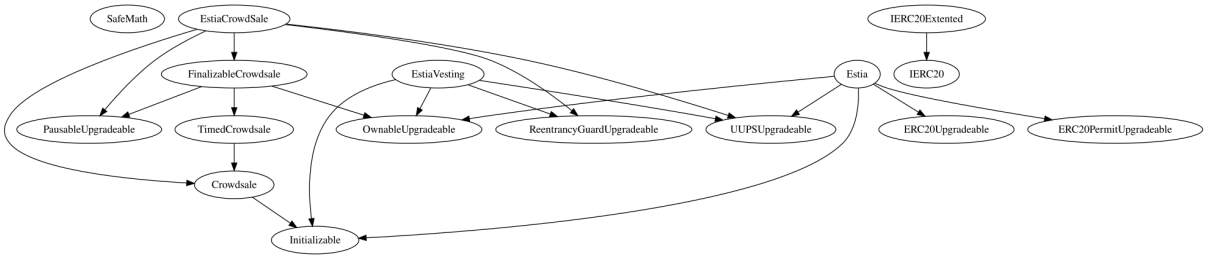
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
EstiaVesting	Implementation	Initializable, OwnableUpgradeable, UUPSUpgradeable, ReentrancyGuardUpgradeable		
		Public	✓	initializer
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	addCrowdsaleAddress	External	✓	onlyOwner
	addTokenGrant	External	✓	nonReentrant onlyCrowdsale
	claimVestedTokens	External	✓	nonReentrant
	getTotalGrantClaimed	External		-
	revokeTokenGrant	External	✓	nonReentrant onlyOwner
	getGrantStartTime	External		-
	getGrantAmount	External		-
	calculateGrantClaim	Public		-
	updateMonthsTime	External	✓	onlyOwner nonReentrant
	currentTime	Private		
	remainingToken	External		-
	nextClaimDate	External		-

	withdrawToken	External	✓	onlyOwner nonReentrant
	withdrawEther	External	✓	onlyOwner nonReentrant
	changeStartTime	External	✓	onlyOwner nonReentrant
	addFounders	External	✓	onlyOwner nonReentrant
	isFounder	External		-
IERC20Extended	Implementation	IERC20		
	decimals	Public		-
Crowdsale	Implementation	Initializable		
	__Crowdsale_init_unchained	Internal	✓	
		External	Payable	-
	buyTokens	Internal	✓	
	_preValidatePurchase	Internal	✓	
	_deliverTokens	Internal	✓	
	_processPurchase	Internal	✓	
	_getTokenAmount	Internal		
	_changeRate	Internal	✓	
	_changeInitialLockInPeriodInSeconds	Internal	✓	
	_changeVestingInMonths	Internal	✓	
	_changeToken	Internal	✓	
	_changeUsdtToken	Internal	✓	

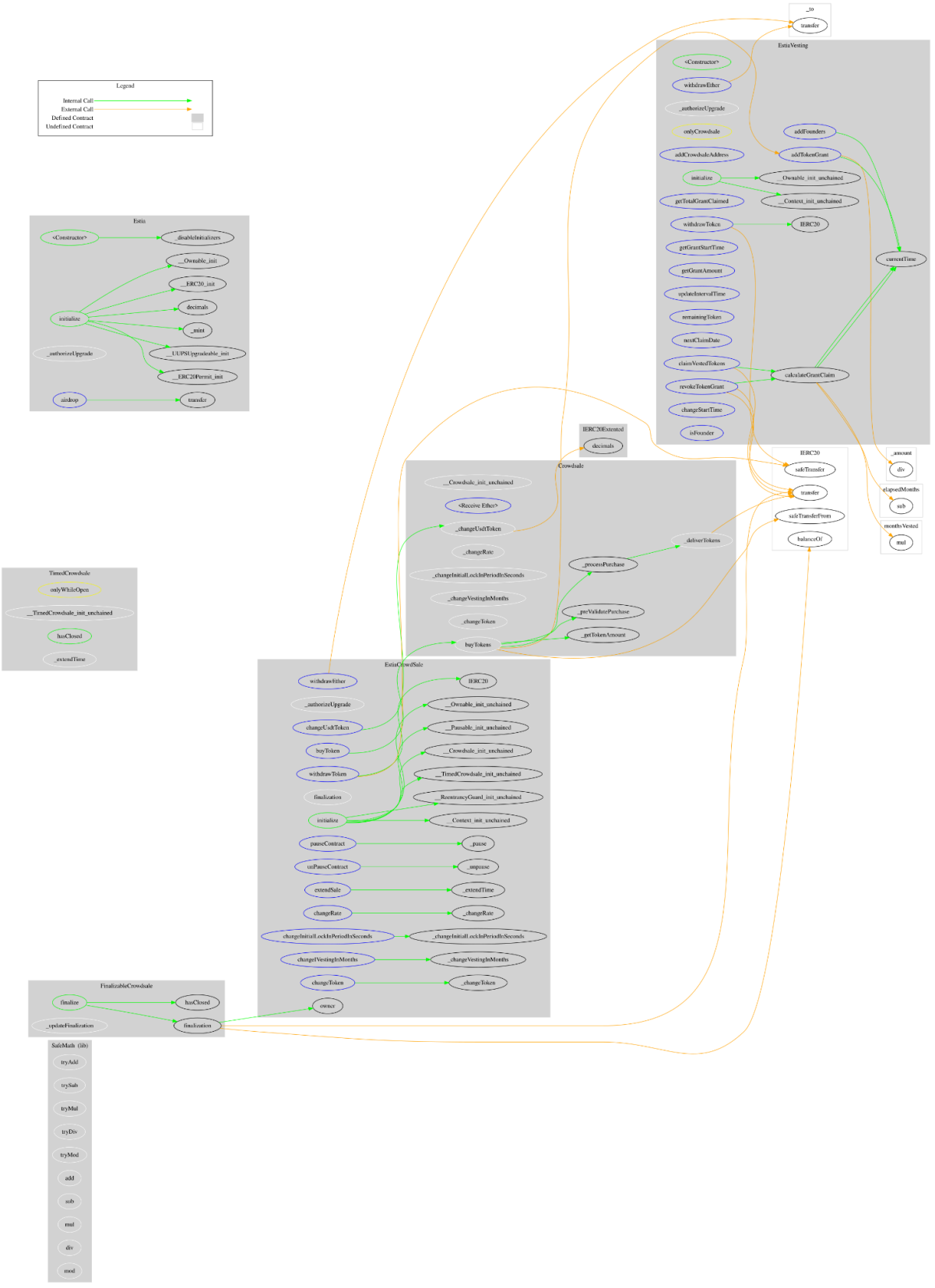
TimedCrowdsale	Implementation	Crowdsale		
	__TimedCrowdsale_init_unchained	Internal	✓	
	hasClosed	Public		-
	_extendTime	Internal	✓	
FinalizableCrowdsale	Implementation	TimedCrowdsale, OwnableUpgradeable, PausableUpgradeable		
	finalize	Public	✓	onlyOwner whenNotPaused
	finalization	Internal	✓	
	_updateFinalization	Internal	✓	
EstiaCrowdSale	Implementation	Crowdsale, PausableUpgradeable, FinalizableCrowdsale, ReentrancyGuardUpgradeable, UUPSUpgradeable		
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	pauseContract	External	✓	onlyOwner
	unPauseContract	External	✓	onlyOwner
	buyToken	External	✓	onlyWhileOpen whenNotPaused nonReentrant

	finalization	Internal	✓	
	extendSale	External	✓	onlyOwner whenNotPaused
	changeRate	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	changeInitialLockInPeriodInSeconds	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	changeIVestingInMonths	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	changeToken	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	changeUsdtToken	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	withdrawToken	External	✓	onlyOwner nonReentrant
	withdrawEther	External	✓	onlyOwner
Estia	Implementation	Initializable, ERC20Upgradeable, OwnableUpgradeable, ERC20PermitUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	airdrop	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Estia
<Constructor>
_disableInitializers
__Ownable_init
__ERC20_init
initialize
decimals
_mint
_authorizeUpgrade
__UUPSUpgradeable_init
__ERC20Permit_init
airdrop
transfer
_to
transfer
EstiaVesting
<Constructor>
withdrawEther
_authorizeUpgrade
onlyCrowdsale
addFounders
addTokenGrant
addCrowdsaleAddress
initialize
__Ownable_init_unchained
getTotalGrantClaimed
__Context_init_unchained
withdrawToken
IERC20
getGrantStartTime
currentTime
getGrantAmount
updateIntervalTime
remainingTokens
nextClaimDate
claimVestedTokens
calculateGrantClaim
revokeTokenGrant
changeStartTime
isFounder
IERC20Extented
decimals
IERC20
safeTransfer
transfer
safeTransferFrom
balanceOf
_amount
div
elapsedMonths
sub
monthsVested
mul
Crowdsale
__Crowdsale_init_unchained
<Receive Ether>
_changeUsdtToken
_changeRate
_changeInitialLockInPeriodInSeconds
_changeVestingInMonths
_changeToken
buyTokens
_deliverTokens
_processPurchase
_preValidatePurchase
_getTokenAmount
TimedCrowdsale
onlyWhileOpen
__TimedCrowdsale_init_unchained
hasClosed
_extendTime
EstiaCrowdSale
withdrawEther
IERC20
_authorizeUpgrade
__Ownable_init_unchained
changeUsdtToken
__Pausable_init_unchained
buyToken
__Crowdsale_init_unchained
withdrawToken
__TimedCrowdsale_init_unchained
finalization
__ReentrancyGuard_init_unchained
initialize
__Context_init_unchained
pauseContract
_pause
unPauseContract
_unpause
extendSale
_extendTime
changeRate
_changeRate
changeInitialLockInPeriodInSeconds
_changeInitialLockInPeriodInSeconds
changeVestingInMonths
_changeVestingInMonths
changeToken
_changeToken
owner
FinalizableCrowdsale
finalize
hasClosed
_updateFinalization
finalization
SafeMath (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
add
sub
mul
div
mod

Summary

The EstiaContracts suite implements a comprehensive token distribution and vesting system, including a crowdsale, vesting schedules, and token management for the Estia token (EST). This audit investigates security vulnerabilities, adherence to best practices, and the overall robustness of the business logic to ensure secure and effective token handling.

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

cyberscope.io