Cyberscope

Audit Report

Estiapayments

September 2024

Repository	https://github.com/Estiapayments/Estia
Commit	e6bd9001a6b7c6e6bc8c0128c74af1f844709721
Audited by	© cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **6**
Review **7**
Audit Updates 7
Source Files 7
Contract Readability Comment **8**
Overview **9**
Estia Contract 9
EstiaCrowdSale Contract 9
Token Purchase Functionality 9
Vesting Mechanism 9
Crowdsale Timing 10
Finalization Functionality 10
Owner Functionalities 10
Withdrawal Functions 10
EstiaVesting Contract 11
Token Grant Functionality 11
Vesting and Claiming Mechanism 11
Revocation of Grants 11
Founders and Special Vesting Rules 11
Grant Management and Update Functions 12
Withdraw Functions 12
USDT Contract 13
Findings Breakdown **14**
Diagnostics **15**
MAC - Missing Access Control 18
Description 18
Recommendation 19
ILE - Infinite Loop Execution 20
Description 20
Recommendation 20
TCI - Token Claim Inconsistency 21
Description 21
Recommendation 22
UGR - Unauthorized Grant Revocation 23
Description 23
Recommendation 23
UVT - Unrestricted Vesting Timeframe 25
Description 25

Recommendation 26
VPTM - Vesting Period Timeframes Misalignment 28
Description 28
Recommendation 30
VUFI - Vesting Update Functionality Inconsistency 31
Description 31
Recommendation 32
ZD - Zero Division 34
Description 34
Recommendation 35
ITD - Inconsistent Token Decimals 36
Description 36
Recommendation 36
UPU - Unutilized Parameter Usage 38
Description 38
Recommendation 40
IVL - Inadequate Vesting Lock 41
Description 41
Recommendation 41
ALM - Array Length Mismatch 42
Description 42
Recommendation 42
CO - Code Optimization 44
Description 44
Recommendation 45
CCS - Commented Code Segments 47
Description 47
Recommendation 47
CCR - Contract Centralization Risk 48
Description 48
Recommendation 50
DPI - Decimals Precision Inconsistency 51
Description 51
Recommendation 51
DTL - Duplicated Transfer Logic 53
Description 53
Recommendation 53
DVAS - Duplicated Vesting Address Set 54
Description 54
Recommendation 54
HLD - Hardcoded Lock Duration 55
Description 55

Recommendation 55
IAVU - Inconsistent addFounders Vesting Update 56
Description 56
Recommendation 56
ITE - Incorrect Time Extension 57
Description 57
Recommendation 57
MCE - Misleading Comment Explanation 58
Description 58
Recommendation 58
MMN - Misleading Method Naming 59
Description 59
Recommendation 59
MUTU - Misleading USDT Token Usage 60
Description 60
Recommendation 60
MC - Missing Check 61
Description 61
Recommendation 62
MCU - Missing CrowdsaleRound Usage 63
Description 63
Recommendation 64
MEM - Missing Error Messages 65
Description 65
Recommendation 65
MEE - Missing Events Emission 66
Description 66
Recommendation 66
MTEE - Missing Transfer Event Emission 67
Description 67
Recommendation 67
MVIC - Missing Vesting Interface Check 68
Description 68
Recommendation 68
RFC - Redundant Function Call 69
Description 69
Recommendation 69
RSML - Redundant SafeMath Library 70
Description 70
Recommendation 70
RTF - Redundant Timestamp Function 71
Description 71

Recommendation 71
RUC - Redundant uint256 Casting 72
Description 72
Recommendation 72
TUU - Time Units Usage 73
Description 73
Recommendation 73
TSI - Tokens Sufficiency Insurance 74
Description 74
Recommendation 74
OCTD - Transfers Contract's Tokens 75
Description 75
Recommendation 75
TIC - Typo in Comments 77
Description 77
Recommendation 77
URL - Unreasonable Revoke Logic 78
Description 78
Recommendation 79
ZAI - Zero Address Initialization 81
Description 81
Recommendation 81
L02 - State Variables could be Declared Constant 83
Description 83
Recommendation 83
L04 - Conformance to Solidity Naming Conventions 84
Description 84
Recommendation 85
L13 - Divide before Multiply Operation 86
Description 86
Recommendation 86
L15 - Local Scope Variable Shadowing 87
Description 87
Recommendation 87
L19 - Stable Compiler Version 88
Description 88
Recommendation 88
L20 - Succeeded Transfer Check 89
Description 89
Recommendation 89
Functions Analysis 90
Inheritance Graph 95

Flow Graph **96**
Summary **97**
Disclaimer **98**
About Cyberscope **99**

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
Commit	e6bd9001a6b7c6e6bc8c0128c74af1f844709721

Audit Updates

Initial Audit	20 Sep 2024

Source Files

Filename	SHA256
USDT.sol	6369716f5c463eccce9e34a84206e6614da46719b7be75447316ed3a04aad8ac
EstiaVesting.sol	6a9281f1ee1156f2f1d87369813499f70d7696b340cb77024a53dc05ab314745
EstiaCrowdSale.sol	daf48e71d0f20e0db9fa2524bd7854612a16a5ee2aee59c134dfd472291a245d
Estia.sol	3bfeddc2aa6b68d089c34d9a7d32b945dd70ad148bd0f2afcf3a21d26ceb1d63

Contract Readability Comment

The audit scope is to check for security vulnerabilities, validate the business logic and propose potential optimizations. The contract is missing the fundamental principles of a Solidity smart contract regarding gas consumption, code readability, and data structures. According to the previously mentioned issues, the contract cannot be assumed that it is in a production-ready state. Given these issues, it is not advisable to assume that the contract is in a production-ready state. The development team is strongly encouraged to re-evaluate the business logic and Solidity guidelines to ensure that the contract adheres to established best practices and security measures. It is recommended that the team review the contract's gas consumption and optimize it accordingly to minimize costs and improve the contract's efficiency. The code's readability should also be improved by simplifying function definitions and using descriptive variable names, as this will enhance the contract's auditability and maintenance.

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 for adding or updating token grants for recipients. Each grant is defined by parameters such as the recipient address, total token amount, initial lock period, vesting duration, and the specific crowdsale round it belongs to. 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

Users 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 and Special Vesting Rules

The contract includes special provisions for founders, defined by the `isFounder` mapping. Founders have a different vesting duration, controlled by the `founderVestingDuration` parameter. The owner can update the status of founders using the `addFounders` function, which adjusts the vesting parameters accordingly. This allows the contract to differentiate between normal recipients and founders, providing tailored vesting schedules for different types of stakeholders.

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

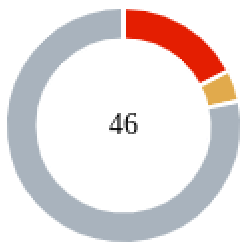
The contract includes secure withdrawal functions for both ERC-20 tokens and Ether. The `withdrawToken` function allows the owner to withdraw any ERC-20 tokens from the contract, while the `withdrawEther` function enables the withdrawal of Ether. These functions ensure that the owner can recover funds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

USDT Contract

The USDT contract represents a basic implementation of the ERC-20 token standard for Tether (USDT), with a fixed total supply of `1000000000000000` tokens. The contract includes standard functionalities such as transferring tokens between addresses, approving allowances for delegated spending, and enabling third-party transfers through the `transferFrom` function. All tokens are initially assigned to the deployer’s address, making them the initial holder of the entire supply.

This contract allows users to securely transfer tokens, set spending allowances, and enable third-party transfers, making it suitable for a wide range of applications within the Ethereum ecosystem, such as trading, payments, and integration with decentralized finance (DeFi) platforms.

Findings Breakdown



●	Critical	8
●	Medium	2
●	Minor / Informative	36

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	8	0	0	0
●	Medium	2	0	0	0
●	Minor / Informative	36	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	MAC	Missing Access Control	Unresolved
🔴	ILE	Infinite Loop Execution	Unresolved
🔴	TCI	Token Claim Inconsistency	Unresolved
🔴	UGR	Unauthorized Grant Revocation	Unresolved
🔴	UVT	Unrestricted Vesting Timeframe	Unresolved
🔴	VPTM	Vesting Period Timeframes Misalignment	Unresolved
🔴	VUFI	Vesting Update Functionality Inconsistency	Unresolved
🔴	ZD	Zero Division	Unresolved
🟠	ITD	Inconsistent Token Decimals	Unresolved
🟠	UPU	Unutilized Parameter Usage	Unresolved
⚪	IVL	Inadequate Vesting Lock	Unresolved
⚪	ALM	Array Length Mismatch	Unresolved
⚪	CO	Code Optimization	Unresolved
⚪	CCS	Commented Code Segments	Unresolved

●	CCR	Contract Centralization Risk	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	DTL	Duplicated Transfer Logic	Unresolved
●	DVAS	Duplicated Vesting Address Set	Unresolved
●	HLD	Hardcoded Lock Duration	Unresolved
●	IAVU	Inconsistent addFounders Vesting Update	Unresolved
●	ITE	Incorrect Time Extension	Unresolved
●	MCE	Misleading Comment Explanation	Unresolved
●	MMN	Misleading Method Naming	Unresolved
●	MUTU	Misleading USDT Token Usage	Unresolved
●	MC	Missing Check	Unresolved
●	MCU	Missing CrowdsaleRound Usage	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MTEE	Missing Transfer Event Emission	Unresolved
●	MVIC	Missing Vesting Interface Check	Unresolved
●	RFC	Redundant Function Call	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved

●	RTF	Redundant Timestamp Function	Unresolved
●	RUC	Redundant uint256 Casting	Unresolved
●	TUU	Time Units Usage	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	TIC	Typo in Comments	Unresolved
●	URL	Unreasonable Revoke Logic	Unresolved
●	ZAI	Zero Address Initialization	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

MAC - Missing Access Control

Criticality	Critical
Location	EstiaCrowdSale.sol#L323
Status	Unresolved

Description

The contract is missing access control on the `addTokenGrant` function, meaning that any user can call this function and allocate tokens for vesting without providing payment or authorization. This exposes the contract to abuse, allowing unauthorized users to manipulate the vesting allocations, which could lead to an incorrect distribution of tokens and harm the integrity of the vesting process. For instance, users have the ability to invoke the `addTokenGrant` function multiple times and then call the `claimVestedTokens` function in order to drain the tokens of the contract, effectively bypassing the intended allocation mechanism.

```
    function addTokenGrant(
            address _recipient,
            uint256 _amount,
            uint256 initialLock, //vesting starts after this holding
perios(in seconds)
            uint256 _vestingDurationInMonths, //10 (in months)
            uint256 _lockDurationInMonths, //1 (in months)
            uint256 crowdsaleRound //1
        ) external nonReentrant {
            ...
                Grant memory grant = Grant({
                    startTime:
currentTime().add(initialLock).add(intervalTime),
                    amount: _amount,
                    vestingDuration: _vestionDuration,
                    monthsClaimed: 0,
                    totalClaimed: 0,
                    recipient: _recipient
                });
                tokenGrants[_recipient][crowdsaleRound] = grant;
                emit GrantAdded(_recipient);
            } else {
                Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
                require(
                    tokenGrant.monthsClaimed < tokenGrant.vestingDuration,
                    "Grant fully claimed"
                );
                tokenGrant.amount = uint256(tokenGrant.amount.add(_amount));
                emit GrantUpdateAmount(_recipient, tokenGrant.amount,
_amount);
            }

        }
```

Recommendation

It is recommended to implement appropriate access control for the `addTokenGrant` function, restricting it to authorized roles such as the contract owner or a designated admin, to ensure that only authorized parties can allocate vesting tokens and maintain the integrity and security of the vesting process. Additionally, it is advisable to restrict calls to the `addTokenGrant` function exclusively to the `EstiaCrowdSale` contract by using a modifier like `onlyCrowdsale`, which verifies that the caller is the `EstiaCrowdSale` contract. This ensures that only the designated crowdsale contract can allocate token grants, further enhancing the security and integrity of the token distribution mechanism.

ILE - Infinite Loop Execution

Criticality	Critical
Location	EstiaVesting.sol#L539
Status	Unresolved

Description

The `nextClaimDate` function contains an infinite loop due to the way the `j` variable is incremented. Since `j` is increased within the loop without proper exit conditions, the loop will run indefinitely in case where the `block.timestamp` equals `finalDate`, causing the contract to become unresponsive and consume excessive gas. This poses a significant risk to the functionality of the contract and could prevent users from retrieving their next claim date.

```
    function nextClaimDate(
        address _recipient,
        uint256 crowdsaleRound
    ) external view returns (uint256) {
        Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
        ...
        uint256 j = 1;
        for (uint i = 0; i < j; i++) {
            startTimeOfUser += intervalTime;
            if (startTimeOfUser > block.timestamp) {
                return startTimeOfUser;
            } else {
                j++;
            }
        }
        return 0;
    }
```

Recommendation

It is recommended to refactor the loop logic by replacing the `for` loop with a `while` loop that has a clear exit condition based on the time comparison, ensuring that the `j` variable is properly managed. This approach will prevent the possibility of an infinite loop, reduce gas consumption, and ensure the function executes efficiently within gas limits.

TCI - Token Claim Inconsistency

Criticality	Critical
Location	EstiaCrowdSale.sol#L93 EstiaVesting.sol#L374
Status	Unresolved

Description

The contract is designed to handle presale purchases and transfer the reward tokens to a vesting contract, allowing users to claim them over time. However, within the vesting contract, the token address can be changed to any other address, creating a scenario where the presale tokens intended for vesting can be replaced with a different token. This can lead to an inconsistency where users who participated in the presale and expect to claim the original reward tokens may end up receiving a different token altogether. This undermines the integrity of the vesting process and the trust of the participants.

```
    function buyTokens(
        address _beneficiary,
        uint256 usdtAmount
    ) internal {
        _preValidatePurchase(_beneficiary, usdtAmount);
        usdtToken.safeTransferFrom(msg.sender, address(this),
usdtAmount);
        // calculate token amount to be created
        uint256 tokens = _getTokenAmount(usdtAmount);

        // update state
        usdtRaised = usdtRaised + usdtAmount;

        UserInfo storage user = users[_beneficiary];
        user.usdtContributed += usdtAmount;
        user.estiaRecieved += tokens;

        _processPurchase(address(vestingToken), tokens);

        vestingToken.addTokenGrant(
            _beneficiary,
            tokens,
            initialLockInPeriodInSeconds,
            vestingMonths,
            1,
            round
        );
        ...
        }
```

```
      function claimVestedTokens(uint256 crowdsaleRound) external
nonReentrant {
        ...
        token.transfer(tokenGrant.recipient, amountVested);
    }
```

Recommendation

It is recommended to implement safeguards that prevent the token address within the vesting contract from being changed once set. This will ensure that the tokens allocated during the presale remain consistent and are the same tokens users can claim, thereby maintaining trust and fulfilling the contract’s expected behavior.

UGR - Unauthorized Grant Revocation

Criticality	Critical
Location	EstiaVesting.sol#L407
Status	Unresolved

Description

The contract contains the `revokeTokenGrant` function, which terminates a token grant and transfers all vested tokens to the specified `_recipient`. However, any user can call this function and pass any recipient's address, allowing them to revoke grants for other users without authorization. This could lead to the unintended deletion of grants and the transfer of tokens back to the `_recipient` addresses without their authorization, posing a significant security risk.

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

        uint256 amountNotVested = (
            tokenGrant.amount.sub(tokenGrant.totalClaimed)
        ).sub(amountVested);

        delete tokenGrants[_recipient][crowdsaleRound];
        ....
        if (amountVested > 0) {
            token.transfer(_recipient, amountVested);
        }
    }
```

Recommendation

It is recommended that the function only allows users to revoke their own grants by checking that the `msg.sender` matches the `_recipient` , ensuring they are only revoking their own grant. This will prevent unauthorized users from deleting or modifying grants that do not belong to them, protecting the integrity of the vesting process.

UVT - Unrestricted Vesting Timeframe

Criticality	Critical
Location	EstiaVesting.sol#L323
Status	Unresolved

Description

The contract is allowing users to specify any vesting duration as a parameter in the `addTokenGrant` function. This lack of restriction enables users to set an artificially short vesting period, allowing them to unlock their vested tokens much faster than intended. This could lead to an unfair advantage for some users, undermining the purpose of the vesting schedule and compromising the contract's intended functionality.

```
 function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this holding
perios(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 _lockDurationInMonths, //1 (in months)
        uint256 crowdsaleRound //1
    ) external nonReentrant {
        // require(tokenGrants[_recipient].amount == 0, "Grant already
exists, must revoke first.");
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_lockDurationInMonths <= 10 * 12, "Lock greater than 10
years");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            uint256 _vestionDuration;
            if(isFounder[_recipient]) {
                _vestionDuration = founderVestingDuration;
            } else {
                _vestionDuration = _vestingDurationInMonths;
            }
            Grant memory grant = Grant({
                startTime:
currentTime().add(initialLock).add(intervalTime),
                amount: _amount,
                vestingDuration: _vestionDuration,
                monthsClaimed: 0,
                totalClaimed: 0,
                recipient: _recipient
            });
            tokenGrants[_recipient][crowdsaleRound] = grant;
            emit GrantAdded(_recipient);
        } else {
            ...
        }
```

Recommendation

It is recommended to enforce minimum vesting duration limits within the `addTokenGrant` function to prevent users from setting unreasonably short vesting periods. This would ensure that the vesting mechanism operates as intended, with users only able to unlock their tokens after a predefined, reasonable timeframe, maintaining the integrity of the vesting process.

VPTM - Vesting Period Timeframes Misalignment

Criticality	Critical
Location	EstiaVesting.sol#L323,374
Status	Unresolved

Description

The contract is using an inaccurate representation of vesting periods, where the intended durations in months are mistakenly applied as seconds in the implementation. The `initialize` and `addTokenGrant` functions include parameters for vesting duration and lock duration expressed in months. However, these timeframes are not correctly converted to seconds before being used in calculations and storage. As a result, the vesting logic is governed by seconds rather than the intended monthly periods. Specifically, the `intervalTime` is set to 180 seconds instead of the correct value for one month, which is 2,592,000 seconds. This discrepancy can lead to unintended release schedules and an inaccurate vesting process for token grants.

```
   function initialize(IERC20 _token) public initializer {
        require(address(_token) != address(0));
        token = _token;
        intervalTime = 180; // in seconds after lockin period user can
able to vest their all tokens in a single shot
        founderVestingDuration = 10; // in months total alloated tokens
divided into this months for founders only.
        ...
    }

      function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this holding
perios(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 _lockDurationInMonths, //1 (in months)
        uint256 crowdsaleRound //1
    ) external nonReentrant {
        // require(tokenGrants[_recipient].amount == 0, "Grant already
exists, must revoke first.");
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_lockDurationInMonths <= 10 * 12, "Lock greater than 10
years");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            uint256 _vestionDuration;
            if(isFounder[_recipient]) {
                _vestionDuration = founderVestingDuration;
            } else {
                _vestionDuration = _vestingDurationInMonths;
            }
            Grant memory grant = Grant({
                startTime:
currentTime().add(initialLock).add(intervalTime),
                amount: _amount,
                vestingDuration: _vestionDuration,
                monthsClaimed: 0,
                totalClaimed: 0,
                recipient: _recipient
            });
            tokenGrants[_recipient][crowdsaleRound] = grant;
            emit GrantAdded(_recipient);
        } else {
            Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
            require(
                tokenGrant.monthsClaimed < tokenGrant.vestingDuration,
                "Grant fully claimed"
            );
            tokenGrant.amount = uint256(tokenGrant.amount.add(_amount));
            emit GrantUpdateAmount(_recipient, tokenGrant.amount,
_amount);
        }
```

Recommendation

It is recommended to update the contract to ensure that the vesting and lock durations are correctly converted from months to seconds before being applied. Utilize Solidity's time units like `days`, `weeks`, and `months` for clarity and accuracy. This will ensure that the vesting functionality aligns with the expected monthly timeframes and avoids confusion in the code implementation.

VUFI - Vesting Update Functionality Inconsistency

Criticality	Critical
Location	EstiaVesting.sol#L323
Status	Unresolved

Description

The contract's `addTokenGrant` function, used for adding or updating token grants, has inconsistencies in handling the vesting parameters. When an existing grant is updated, key parameters such as `startTime` and `vestingDuration` are not modified. This allows users to initially vest a small amount and later add a larger amount, while the vesting schedule for the larger amount remains tied to the original `startTime` and `vestingDuration`. This can result in users being able to vest the updated, larger amount based on the earlier vesting period, leading to unfair token distribution and potential exploitation of the vesting mechanism.

```
   function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this holding
perios(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 _lockDurationInMonths, //1 (in months)
        uint256 crowdsaleRound //1
    ) external nonReentrant {
        // require(tokenGrants[_recipient].amount == 0, "Grant already
exists, must revoke first.");
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_lockDurationInMonths <= 10 * 12, "Lock greater than 10
years");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            uint256 _vestionDuration;
            if(isFounder[_recipient]) {
                _vestionDuration = founderVestingDuration;
            } else {
                _vestionDuration = _vestingDurationInMonths;
            }
            Grant memory grant = Grant({
                startTime:
currentTime().add(initialLock).add(intervalTime),
                ...
            });
            tokenGrants[_recipient][crowdsaleRound] = grant;
        else {
            Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
            require(
                tokenGrant.monthsClaimed < tokenGrant.vestingDuration,
                "Grant fully claimed"
            );
            tokenGrant.amount = uint256(tokenGrant.amount.add(_amount));
            emit GrantUpdateAmount(_recipient, tokenGrant.amount,
_amount);
        }
```

Recommendation

It is recommended to update all relevant fields in the vesting schedule, including `startTime` , `vestingDuration` , and any other associated parameters, when a grant is modified. Additionally, consider using separate entries or a mechanism that clearly distinguishes between different grant additions for the same user. This will ensure that each updated grant amount has its own distinct vesting schedule based on the time of the update, thereby preventing users from circumventing the intended vesting process and maintaining fairness in token distribution.

ZD - Zero Division

Criticality	Critical
Location	EstiaVesting.sol#L327
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

Specifically, the `_vestingDurationInMonths` variable can be set to zero.

```
   function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this holding
perios(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 _lockDurationInMonths, //1 (in months)
        uint256 crowdsaleRound //1
    ) external nonReentrant {
        // require(tokenGrants[_recipient].amount == 0, "Grant already
exists, must revoke first.");
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_lockDurationInMonths <= 10 * 12, "Lock greater than 10
years");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            uint256 _vestionDuration;
            if(isFounder[_recipient]) {
                _vestionDuration = founderVestingDuration;
            } else {
                _vestionDuration = _vestingDurationInMonths;
            }
            ....
```

Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

ITD - Inconsistent Token Decimals

Criticality	Medium
Location	EstiaCrowdSale.sol#L462,476
Status	Unresolved

Description

The contract allows the owner to set new contract addresses for the tokens used during the presale, such as the primary token and USDT. However, there is no verification that the new tokens have the same decimal precision as the original tokens. If a token with different decimal precision is introduced, the contract's calculations, such as token amounts and pricing, will be incorrect, leading to significant discrepancies and errors in the presale process.

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

    /**
     * @dev Change the usdt token address.
     * @param _usdtToken The new token address to be used.
     */
    function changeUsdtToken(
        IERC20 _usdtToken
    ) external virtual onlyOwner onlyWhileOpen whenNotPaused {
        _changeUsdtToken(_usdtToken);
    }
```

Recommendation

It is recommended to standardize the decimal precision across all tokens or implement a check to ensure that any new token set by the owner has the same decimal precision as the original token. This will ensure that calculations are accurate and prevent potential issues related to mismatched token decimals during the presale.

UPU - Unutilized Parameter Usage

Criticality	Medium
Location	EstiaVesting.sol#L323
Status	Unresolved

Description

The contract is taking the `_lockDurationInMonths` parameter in the `addTokenGrant` function. However, this parameter is never utilized throughout the code, making it redundant. The presence of an unused parameter adds unnecessary complexity to the function and can confuse users about its intended purpose.

```
    * @param _lockDurationInMonths The lock duration (in months) before
tokens are transferable.
     * @param crowdsaleRound The specific round of the crowdsale to
which this grant belongs.
     */
    function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this holding
perios(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 _lockDurationInMonths, //1 (in months)
        uint256 crowdsaleRound //1
    ) external nonReentrant {
        // require(tokenGrants[_recipient].amount == 0, "Grant already
exists, must revoke first.");
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_lockDurationInMonths <= 10 * 12, "Lock greater than 10
years");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            uint256 _vestionDuration;
            if(isFounder[_recipient]) {
                _vestionDuration = founderVestingDuration;
            } else {
                _vestionDuration = _vestingDurationInMonths;
            }
            Grant memory grant = Grant({
                startTime:
currentTime().add(initialLock).add(intervalTime),
                amount: _amount,
                vestingDuration: _vestionDuration,
                monthsClaimed: 0,
                totalClaimed: 0,
                recipient: _recipient
            });
            tokenGrants[_recipient][crowdsaleRound] = grant;
            emit GrantAdded(_recipient);
        } else {
            Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
            require(
                tokenGrant.monthsClaimed < tokenGrant.vestingDuration,
                "Grant fully claimed"
            );
            tokenGrant.amount = uint256(tokenGrant.amount.add(_amount));
            emit GrantUpdateAmount(_recipient, tokenGrant.amount,
_amount);
        }

        // token.approve(_recipient, address(this), _amount);
        // Transfer the grant tokens under the control of the vesting
contract
        // token.transferFrom(_recipient, address(this), _amount);
    }
```

Recommendation

It is recommended to either remove the `_lockDurationInMonths` parameter if it is not needed, or ensure that it is properly integrated into the function logic. This will help streamline the function, reduce confusion, and ensure clarity about the role of each parameter in the contract.

IVL - Inadequate Vesting Lock

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L357
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;
        vestingAddress = address(vesting);
        vestingMonths = 1;
        initialLockInPeriodInSeconds = 300;
        ...
```

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.

ALM - Array Length Mismatch

Criticality	Minor / Informative
Location	EstiaVesting.sol#L630
Status	Unresolved

Description

The contract is designed to handle the process of elements from arrays through functions that accept multiple arrays as input parameters. These functions are intended to iterate over the arrays, processing elements from each array in a coordinated manner. However, there are no explicit checks to verify that the lengths of these input arrays are equal. This lack of validation could lead to scenarios where the arrays have differing lengths, potentially causing out-of-bounds access if the function attempts to process beyond the end of the shorter array. Such situations could result in unexpected behavior or errors during the contract's execution, compromising its reliability and security.

Specifically, the `_totalVestingMonths` array is not checked to have the same length as the `_recipient` and `startTime` arrays.

```
    function changeStartTime(
        address[] calldata _recipient,
        uint256[] calldata _startTime,
        uint256[] calldata _totalVestingMonths,
        uint256 crowdsaleRound
    ) external onlyOwner nonReentrant {
        require(
            _recipient.length == _startTime.length,
            "Invalid parameters"
        );
        for (uint256 index = 0; index < _recipient.length; index++) {
            require(
            ...
            tokenGrant.startTime = _startTime[index];
            tokenGrant.vestingDuration = _totalVestingMonths[index];
        }
    }
```

Recommendation

To mitigate this, it is recommended to incorporate a validation check at the beginning of the function that accepts multiple arrays to ensure that the lengths of these arrays are identical. This can be achieved by implementing a conditional statement that compares the lengths of the arrays, and reverts the transaction if the lengths do not match. Such a validation step will prevent out-of-bounds errors and ensure that elements from each array are processed in a paired and coordinated manner, thus preserving the integrity and intended functionality of the contract.

CO - Code Optimization

Criticality	Minor / Informative
Location	EstiaVesting.sol#L456
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, the contract is calculating the `amountVestedPerMonth` value within the `calculateGrantClaim` function using a formula that subtracts the `totalClaimed` tokens and `monthsClaimed` months from the total amount and vesting duration, respectively. This calculation is performed every time the function is called, leading to unnecessary computation. Since the `amountVestedPerMonth` can be derived directly as `tokenGrant.amount / tokenGrant.vestingDuration` (assuming consistent monthly vesting), this repeated calculation introduces inefficiency and complexity.

```
function calculateGrantClaim(
        address _recipient,
        uint256 crowdsaleRound
    ) public view returns (uint256, uint256) {
        Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];

       ...
        uint256 elapsedMonths = currentTime()
            .sub(tokenGrant.startTime)
            .div(intervalTime)
            .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);
        } else {
            uint256 monthsVested = uint256(
                elapsedMonths.sub(tokenGrant.monthsClaimed)
            );
            uint256 amountVestedPerMonth = (
                tokenGrant.amount.sub(tokenGrant.totalClaimed)
            ).div(
                    uint256(

tokenGrant.vestingDuration.sub(tokenGrant.monthsClaimed)
                    )
                );
            uint256 amountVested = uint256(
                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.

Recommendation:

It is recommended to precompute the `amountVestedPerMonth` as `tokenGrant.amount / tokenGrant.vestingDuration` when the grant is created and store this value in the `tokenGrant` struct. This optimization would reduce computational overhead and simplify the logic within the `calculateGrantClaim` function, improving gas efficiency and code readability.

CCS - Commented Code Segments

Criticality	Minor / Informative
Location	EstiaVesting.sol#L331,368
Status	Unresolved

Description

The contract contains several code segments that are commented out. Blocks of code, including important operations and validation checks, are present but commented out. Commented code can be a source of confusion, as it's unclear whether these segments are meant for future use, are remnants of previous iterations, or are temporarily disabled for testing purposes. Moreover, commented out code can clutter the contract, making it more challenging to read and understand the actual functioning code.

```
// require(tokenGrants[_recipient].amount == 0, "Grant already exists,
must revoke first.");
...
// token.approve(_recipient, address(this), _amount);
// Transfer the grant tokens under the control of the vesting contract
// token.transferFrom(_recipient, address(this), _amount);
```

Recommendation

t is recommended to either remove the parts of the code that are not intended to be used or to declare and code the appropriate segments properly if they are meant for future implementation. If the intention is to preserve these segments for historical or reference purposes, it would be beneficial to move them to documentation outside of the active codebase. This approach helps maintain the clarity and cleanliness of the contract's code, ensuring that it accurately reflects its current functionality and intended use.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Estia.sol#L51 EstiaCrowdSale.sol#L453,462,476 EstiaVesting.sol#L630,674
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.

```
    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 changeStartTime(
        address[] calldata _recipient,
        uint256[] calldata _startTime,
        uint256[] calldata _totalVestingMonths,
        uint256 crowdsaleRound
    ) external onlyOwner nonReentrant {
        ...

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


    function addFounders(address[] calldata _founders, bool[] calldata
_status) external onlyOwner nonReentrant {
        require(_founders.length == _status.length, "Invalid
parameters");
        for (uint256 index = 0; index < _founders.length; index++) {
            require(
                _founders[index] != address(0),
                "Invalid founder address"
            );
            isFounder[_founders[index]] = _status[index];
        }

        ...
    }
```

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#L173
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; // rate accepts upto
two decimals that's why 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.

DTL - Duplicated Transfer Logic

Criticality	Minor / Informative
Location	USDT.sol#L21,37
Status	Unresolved

Description

The contract is using both the `transfer` and `transferFrom` functions, which share similar code segments for validating addresses, balances, and updating balances. This duplication of logic increases the potential for errors and makes the contract more difficult to maintain. Instead of repeating similar code in both functions, a separate `_transfer` function could be utilized to handle the shared logic, reducing redundancy and improving code clarity.

```
    function transfer(address to, uint256 value) public returns (bool) {
        require(to != address(0), "Invalid address");
        require(balanceOf[msg.sender] >= value, "Insufficient balance");

        balanceOf[msg.sender] -= value;
        balanceOf[to] += value;
        emit Transfer(msg.sender, to, value);
        return true;
    }

    function transferFrom(address from, address to, uint256 value)
public returns (bool) {
        ...
    }
```

Recommendation

It is recommended to refactor the contract by introducing a private or internal `_transfer` function that encapsulates the common logic used in both `transfer` and `transferFrom`. This would reduce code duplication, improve maintainability, and ensure consistency in how transfers are handled across the contract.

DVAS - Duplicated Vesting Address Set

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L366
Status	Unresolved

Description

The contract is assigning the same vesting contract instance to two different variables, `vestingToken` and `vestingAddress`. This duplication is unnecessary and can introduce confusion for developers or users, as it creates multiple points of reference for the same contract. It increases the risk of errors and complicates contract management, especially if only one of the variables is used or updated while the other remains unused or outdated.

```
    function initialize(
        uint256 rate,
        IERC20 _token,
        IERC20 _usdtToken,
        uint256 openingTime,
        uint256 closingTime,
        EstiaVesting vesting // the vesting contract
    ) public initializer {
        round = 1;
        vestingToken = vesting;
        vestingAddress = address(vesting);
    ...
```

Recommendation

It is recommended to utilize only one variable for storing the vesting contract address, ideally keeping the `vestingAddress` variable for clarity. This will reduce redundancy, prevent confusion, and ensure consistency in how the vesting contract is referenced and used throughout the contract's functions.

HLD - Hardcoded Lock Duration

Criticality	Minor / Informative
Location	EstiaVesting.sol#L111
Status	Unresolved

Description

The contract is using a hardcoded value of `1` for the `_lockDurationInMonths` variable in the `addTokenGrant` function. As a result, this value cannot be modified dynamically based on different vesting requirements. This hardcoding reduces the contract's flexibility and may not accommodate future changes or different vesting schedules that might be required for various beneficiaries or scenarios.

```
vestingToken.addTokenGrant(
    _beneficiary,
    tokens,
    initialLockInPeriodInSeconds,
    vestingMonths,
    1,
    round
);
```

Recommendation

It is recommended to either remove the `_lockDurationInMonths` parameter if the value `1` is always intended to be used or to allow the contract to dynamically set this value based on specific vesting needs. This change will improve the contract's adaptability and ensure it can accommodate varying lock durations in the future.

IAVU - Inconsistent addFounders Vesting Update

Criticality	Minor / Informative
Location	EstiaVesting.sol#L674
Status	Unresolved

Description

The contract is using the `addFounders` function to assign or revoke the founder status for users, which results in different vesting periods for founders compared to regular users. However, when a user's founder status is changed, the current vesting period is not updated accordingly. This can lead to inconsistent vesting schedules, where the vesting period remains set to the previous status, potentially causing unexpected behavior in the token vesting process.

```
   function addFounders(address[] calldata _founders, bool[] calldata
_status) external onlyOwner nonReentrant {
        require(_founders.length == _status.length, "Invalid
parameters");
        for (uint256 index = 0; index < _founders.length; index++) {
            require(
                _founders[index] != address(0),
                "Invalid founder address"
            );
            isFounder[_founders[index]] = _status[index];
        }

        // if founder turn to normal user means need to update his
startVesting like normal user flow & viceversa.
        emit AddFounders(_founders, _status);
    }
```

Recommendation

It is recommended to implement a mechanism that updates the vesting period dynamically when a user's founder status is modified. This would ensure that any changes in status reflect the correct vesting period immediately, maintaining consistency and preventing any unintended vesting discrepancies.

ITE - Incorrect Time Extension

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L287
Status	Unresolved

Description

The contract is utilizing the `_extendTime` function, which is intended to extend the `closingTime` of the crowdsale. However, instead of adding additional time to the existing `closingTime`, the function sets the time to a specific value passed as `newClosingTime`. This approach can lead to a scenario where the new closing time is less than or equal to the current `closingTime`, thus failing to extend the crowdsale duration as intended.

```
    /**
     * @dev Extend crowdsale.
     * @param newClosingTime Crowdsale closing time
     */
    function _extendTime(uint256 newClosingTime) internal {
        require(
            newClosingTime >= openingTime,
            "Closing time cant be before opening time"
        );
        closingTime = newClosingTime;
        emit TimedCrowdsaleExtended(closingTime, newClosingTime);
    }
```

Recommendation

It is recommended to refactor the `_extendTime` function so that the `newClosingTime` is added to the existing `closingTime` value. This will ensure that the function truly extends the crowdsale duration, reflecting an actual extension rather than simply resetting the closing time to a specific value, which may unintentionally shorten the sale period.

MCE - Misleading Comment Explanation

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L176
Status	Unresolved

Description

The contract is using a comment that states the `rate` variable "accepts up to two decimals" in the calculation of the `tRate`. However, the `rate` variable itself can be set to any number, not limited to two decimal places, which makes the comment misleading. This could cause confusion for developers and users interacting with the code, leading to potential misinterpretation of the contract’s behavior or logic.

```
uint256 tRate = (rate * 10 ** 6) / 10000; // rate accepts upto two
decimals that's why 10000
```

Recommendation

It is recommended to update the comment to accurately reflect the behavior of the `rate` variable, clarifying that the rate is not inherently limited to two decimal places. This will ensure that anyone using the code understands the full flexibility of the `rate` variable and prevent any confusion or misinterpretation regarding how the rate is used in calculations.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L415
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.

MUTU - Misleading USDT Token Usage

Criticality	Minor / Informative
Location	USDT.sol#L5
Status	Unresolved

Description

The contract is declaring a new token with the name "USDT (Tether)" and the symbol "USDT." This is misleading because the USDT (Tether) token already exists as a widely recognized stablecoin on multiple blockchains. Deploying a token with the same name and symbol can cause confusion among users, leading them to believe they are interacting with the established USDT token rather than a new, unrelated token. This can also create potential issues when interfacing with third-party applications, exchanges, and wallets that recognize the original USDT token.

```
contract USDT {
    string public name = "USDT (Tether)";
    string public symbol = "USDT";
    uint8 public decimals = 6;
    uint256 public totalSupply = 1000000000000000 * 10 ** 6;
```

Recommendation

It is recommended to use the original USDT contract address if interaction with USDT is required. Alternatively, if this token is intended to be distinct, it is recommended to choose a different name and symbol to avoid confusion and ensure clarity for users and external systems interacting with the token.

MC - Missing Check

Criticality	Minor / Informative
Location	EstiaVesting.sol#L323
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The `addTokenGrant` function is missing a check to verify that the `_recipient` address is not set to the zero address.

```
 function addTokenGrant(
        address _recipient,
        ...
    ) external nonReentrant {
        ...
        if (tokenGrants[_recipient][crowdsaleRound].amount == 0) {
            uint256 _vestionDuration;
            if(isFounder[_recipient]) {
                _vestionDuration = founderVestingDuration;
            } else {
                _vestionDuration = _vestingDurationInMonths;
            }
            Grant memory grant = Grant({
                startTime:
currentTime().add(initialLock).add(intervalTime),
                amount: _amount,
                vestingDuration: _vestionDuration,
                monthsClaimed: 0,
                totalClaimed: 0,
                recipient: _recipient
            });
            tokenGrants[_recipient][crowdsaleRound] = grant;
            emit GrantAdded(_recipient);
        } else {
            Grant storage tokenGrant =
tokenGrants[_recipient][crowdsaleRound];
            ...
            emit GrantUpdateAmount(_recipient, tokenGrant.amount,
_amount);
        }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MCU - Missing CrowdsaleRound Usage

Criticality	Minor / Informative
Location	EstiaVesting.sol#L323
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.

```
    * @param crowdsaleRound The specific round of the crowdsale to which
this grant belongs.
     */
    function addTokenGrant(
        address _recipient,
        uint256 _amount,
        uint256 initialLock, //vesting starts after this holding
perios(in seconds)
        uint256 _vestingDurationInMonths, //10 (in months)
        uint256 _lockDurationInMonths, //1 (in months)
        uint256 crowdsaleRound //1
    ) external nonReentrant {
        // require(tokenGrants[_recipient].amount == 0, "Grant already
exists, must revoke first.");
        require(
            _vestingDurationInMonths <= 25 * 12,
            "Duration greater than 25 years"
        );
        require(_lockDurationInMonths <= 10 * 12, "Lock greater than 10
years");
        require(_amount != 0, "Grant amount cannot be 0");
        uint256 amountVestedPerMonth =
_amount.div(_vestingDurationInMonths);
        require(amountVestedPerMonth > 0, "amountVestedPerMonth < 0");

        if (tokenGrants[_recipient][crowdsaleRound].amount == 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.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	EstiaVesting.sol#L294
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(address(_token) != address(0))
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	EstiaVesting.sol#L630
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 changeStartTime(
        address[] calldata _recipient,
        uint256[] calldata _startTime,
        uint256[] calldata _totalVestingMonths,
        uint256 crowdsaleRound
    ) external onlyOwner nonReentrant {
       ...

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

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.

MTEE - Missing Transfer Event Emission

Criticality	Minor / Informative
Location	USDT.sol#L18
Status	Unresolved

Description

The contract does not emit an event when portions of the main amount are transferred during the transfer process. This lack of event emission results in decreased transparency and traceability regarding the flow of tokens, and hinders the ability of decentralized applications (dApps), such as blockchain explorers, to accurately track and analyze these transactions.

```
balanceOf[msg.sender] = totalSupply;
```

Recommendation

It is advisable to incorporate the emission of detailed event logs following each asset transfer. These logs should encapsulate key transaction details, including the identities of the sender and receiver, and the quantity of assets transferred. Implementing this practice will enhance the reliability and transparency of transaction tracking systems, ensuring accurate data availability for ecosystem participants.

MVIC - Missing Vesting Interface Check

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L462
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.

RFC - Redundant Function Call

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L333,430
Status	Unresolved

Description

The contract is making a redundant call to the `_updateFinalization` function within the `extendSale` function. Since the `isFinalized` variable is already checked and confirmed to be `false` before calling `_updateFinalization`, setting `isFinalized = false` again serves no practical purpose. This redundancy adds unnecessary complexity to the function without providing any additional value.

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

    function extendSale(
        uint256 newClosingTime
    ) external virtual onlyOwner whenNotPaused {
        require(!isFinalized, "Sale Finalized");
        _extendTime(newClosingTime);
        _updateFinalization();
    }
```

Recommendation

It is recommended to remove the `_updateFinalization` function call from the `extendSale` function. This will streamline the code by eliminating redundant logic, making the function more efficient and easier to understand while maintaining the intended functionality.

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.

RTF - Redundant Timestamp Function

Criticality	Minor / Informative
Location	EstiaVesting.sol#L468
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#L364,384
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.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L368 EstiaCrowdSale.sol#L296
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
vestingMonths = 1;
initialLockInPeriodInSeconds = 300;
```

```
intervalTime = 180; // in seconds after lockin period user can able to
vest their all tokens in a single shot
founderVestingDuration = 10;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds`, `minutes`, `hours`, `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	EstiaCrowdSale.sol#L149
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#L488
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.

TIC - Typo in Comments

Criticality	Minor / Informative
Location	EstiaVesting.sol#L326
Status	Unresolved

Description

The contract contains a typo in the comment section, where the word "perios" is used instead of "periods" in the comment `// vesting starts after this holding perios(in seconds)`. This typo may not impact the functionality of the contract but can cause confusion for developers and users who are using the code, potentially leading to misunderstandings about the intended functionality.

```
uint256 initialLock, //vesting starts after this holding perios(in
seconds)
```

Recommendation

It is recommended to correct the typo in the comment by replacing "perios" with "periods" to improve the clarity and readability of the code. Ensuring accurate comments helps maintain code quality and facilitates better understanding for future developers and users.

URL - Unreasonable Revoke Logic

Criticality	Minor / Informative
Location	EstiaVesting.sol#L374,407
Status	Unresolved

Description

The contract contains two functions, `claimVestedTokens` and `revokeTokenGrant` , which allow users to claim their vested tokens. However, the `revokeTokenGrant` function performs the same functionality as `claimVestedTokens` but deletes the remaining grant, causing the user to lose any unvested tokens. Consequently, there is no incentive for users to use the `revokeTokenGrant` function, as they can achieve the same result by using the `claimVestedTokens` function and waiting for future vesting periods without forfeiting any tokens.

```
function claimVestedTokens(uint256 crowdsaleRound) external nonReentrant
{
        uint256 monthsVested;
        uint256 amountVested;
        (monthsVested, amountVested) = calculateGrantClaim(
            msg.sender,
            crowdsaleRound
        );
        ...
        token.transfer(tokenGrant.recipient, amountVested);
    }

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

        uint256 amountNotVested = (
            tokenGrant.amount.sub(tokenGrant.totalClaimed)
        ).sub(amountVested);

        delete tokenGrants[_recipient][crowdsaleRound];

        emit GrantRevoked(_recipient, amountVested, amountNotVested);

        // only transfer tokens if amounts are non-zero.
        // Negative cases are covered by upperbound check in
addTokenGrant and overflow protection using SafeMath
        if (amountNotVested > 0) {
            token.transfer(crowdsale_address, amountNotVested);
        }
        if (amountVested > 0) {
            token.transfer(_recipient, amountVested);
        }
    }
```

Recommendation

It is recommended to reconsider the intended functionality of the `revokeTokenGrant` function. Refactor its logic to provide users with a tangible benefit or specific use case that incentivizes its usage. This may include adding an option for users to voluntarily forfeit their unvested tokens in exchange for a reward or modifying the function to serve a distinct purpose not covered by `claimVestedTokens` .

ZAI - Zero Address Initialization

Criticality	Minor / Informative
Location	EstiaVesting.sol#L284,304
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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	USDT.sol#L6,7,8,9
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
string public name = "USDT (Tether)"
string public symbol = "USDT"
uint8 public decimals = 6
uint256 public totalSupply = 1000000000000000 * 10 ** 6
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	EstiaVesting.sol#L284,293,324,325,327,328,396,408,438,446,457,512,528,540,588,589,603,604,631,632,633,674 EstiaCrowdSale.sol#L69,70,71,94,256,257,258,359,360,406,477,489,490,504,505
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
uint256 _lockDurationInMonths
uint256 _intervalTime
address _tokenContract
address payable _to
address[] calldata _recipient
uint256[] calldata _startTime
uint256[] calldata _totalVestingMonths
address[] calldata _founders
bool[] calldata _status

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	EstiaVesting.sol#L494,501 EstiaCrowdSale.sol#L176,177,178
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(
                    uint256(

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

uint256 tokens = (_usdtAmount * tRate) / 10 ** 6
return tokens * 10 ** 12
```

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#L358,361,362
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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	USDT.sol#L3 EstiaVesting.sol#L2 EstiaCrowdSale.sol#L3 Estia.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 ^0.8.19;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	EstiaVesting.sol#L392,430,433 EstiaCrowdSale.sol#L153,418
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**
USDT	Implementation			
		Public	✓	-
	transfer	Public	✓	-
	approve	Public	✓	-
	transferFrom	Public	✓	-
EstiaVesting	Implementation	Initializable, OwnableUpgradeable, UUPSUpgradeable, ReentrancyGuardUpgradeable		
		Public	✓	initializer
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	addCrowdsaleAddress	External	✓	onlyOwner
	addTokenGrant	External	✓	nonReentrant
	claimVestedTokens	External	✓	nonReentrant
	getTotalGrantClaimed	External		-
	revokeTokenGrant	External	✓	nonReentrant
	getGrantStartTime	External		-
	getGrantAmount	External		-

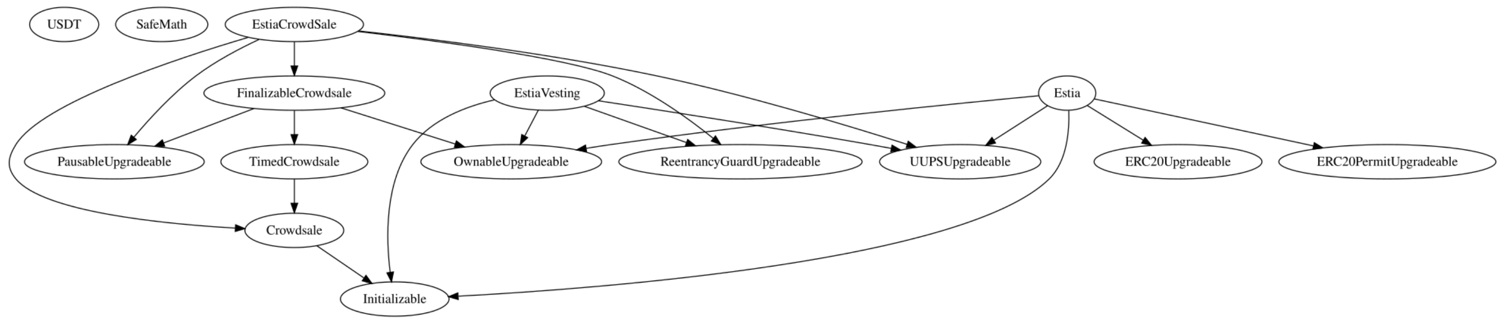
	calculateGrantClaim	Public		-
	updateIntervalTime	External	✓	onlyOwner nonReentrant
	currentTime	Private		
	remainingToken	External		-
	nextClaimDate	External		-
	changeToken	External	✓	onlyOwner nonReentrant
	withdrawToken	External	✓	onlyOwner nonReentrant
	withdrawEther	External	✓	onlyOwner nonReentrant
	changeStartTime	External	✓	onlyOwner nonReentrant
	addFounders	External	✓	onlyOwner nonReentrant
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

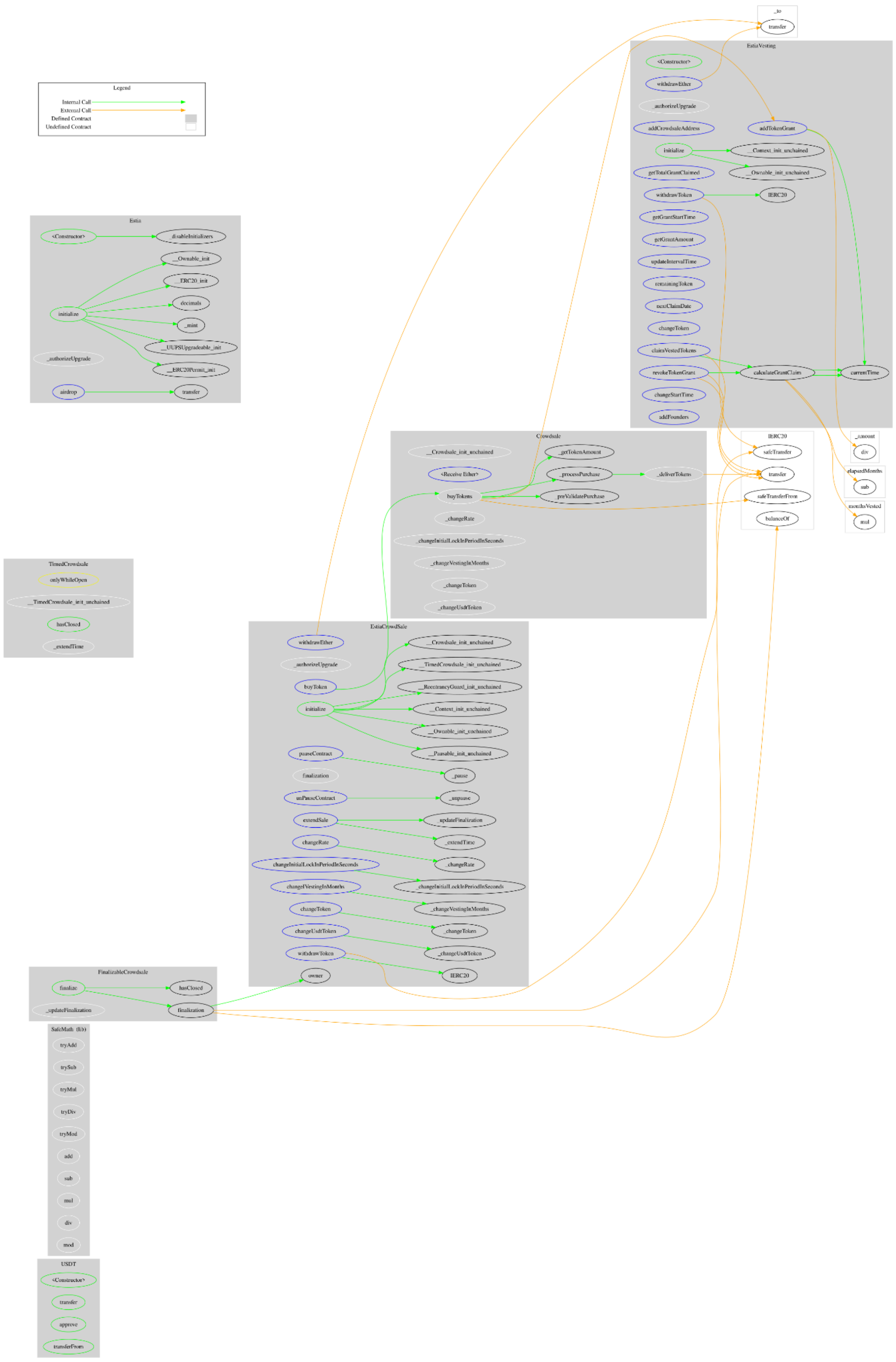
	buyToken	External	✓	onlyWhileOpen whenNotPaused nonReentrant
	finalization	Internal	✓	
	extendSale	External	✓	onlyOwner whenNotPaused
	changeRate	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	changeInitialLockInPeriodInSeconds	External	✓	onlyOwner onlyWhileOpen whenNotPaused
	changeIVestingInMonths	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
EstiaVesting
Crowdsale
EstiaCrowdSale
TimedCrowdsale
FinalizableCrowdsale
SafeMath (lib)
USDT
IERC20
_to

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