Cyberscope

Audit Report

eXchange1

April 2025

Network BSC, ETH

Audited by © cyberscope

Table of Contents

Risk Classification

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

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

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

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

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

Review

Deployments

Contract	Contract Name	Proxy Address	Implementation
ex1Token.sol	EX1	0xf861e69ebc5b5a66bdb2b464b4fcb122d018a075	0x582bae52f616fb3966989a5fa5ab71ddac1085bc
ex1ICOv2.sol	Ex1ICO	0x13a1d8780aaa9e892edf065577435a34c7d91771	0x57b3a5bc0a5cec7add60c20354a94123d35808a9
ex1ICOVesting.sol	ICOVesting	0xbe745f3c2a7cbd7b7c164f523fe6ca746b58465f	0xfd1f71c85080c82d73c8da876b8ecc9d5c7c1952
ex1Staking.sol	Ex1Staking	0x9e91c92b18d8876679059c1a2f8200d20466f2c3	0xc7ddd75275895f0d96fd7b253007582c0e3e202c
ex1PrivateVesting.sol	PrivateVesting	0xa1424cf8b5f8828384c78a1e1d65d3ced047c961	0x03ea1d67420c4f42861ffa6bdd8184b0e1cfce45
ex1EthICO.sol	Ex1ICO	0x0d7db2c9be6e563b47292de154e7db190db96d0e	0xb39a604fb8fb8c9b79d803e9a435b5b87156dc9d

Audit Updates

Initial Audit	18 Apr 2025

Source Files

Filename	SHA256
ex1Token.sol	00bb4f29d4e3292861c1c2739c1730936d28ce97e122abb8e80d97ac42855fcd

ex1Staking.sol	67fbacb6a42db8a0bb7f511a8c3514be1cfa1621bd7115a615da5cda934dff5e
ex1PrivateVesting.sol	e1320d7f34ef04e4e701b9458d07f687489f35224720a35666c2b00569dedf32
ex1ICOv2.sol	61954a261379cb44d0de30b427529b7582fcdce9279435b92efdd31a7b3cc313
ex1ICOVesting.sol	1deb4b755c002f4e3688638f94c9f3f63f38006dd8eba69012cc14c7f82fa506
ex1EthICO.sol	9eb6a510b9939335493bd9d4f93110a7563c504eeefb431e22308bbae3cedb98

Overview

The suite of six smart contracts, EX1, Ex1ICO (BNB Chain), Ex1ICO (ETH), ICOVesting, Ex1Staking, and PrivateVesting, collectively forms a comprehensive ecosystem for managing the issuance, sale, vesting, staking, and private allocation of the EX1 token, ensuring secure, transparent, and controlled token distribution. Built with OpenZeppelin libraries, the EX1 contract is a multi-signature ERC20 token with upgradeable functionality, enforcing restricted address transfers through a multi-approval process. The Ex1ICO contracts facilitate token sales on BNB Chain (supporting USDC, USDT, ETH, BTC) and Ethereum (ETH only, bridging to BNB Chain), with customizable ICO stages, price feeds, and purchase limits. The ICOVesting contract manages linear vesting of ICO-purchased tokens, allowing periodic claims, while the Ex1Staking contract incentivizes long-term holding by enabling users to stake ICO tokens for dynamic rewards before vesting begins. The PrivateVesting contract handles linear vesting for non-ICO participants, with flexible schedules and revocation options. Together, these contracts provide a robust, role-based, and upgradeable framework for token management, cross-chain sales, and incentivized holding, tailored for both public and private stakeholders.

ex1Token file

The EX1 contract is a multi-signature ERC20 token contract with upgradeable functionality, designed to ensure secure and controlled token transfers through a robust governance framework. Built on OpenZeppelin's Initializable, ERC20Upgradeable, AccessControlUpgradeable, and UUPSUpgradeable standards, it implements a multi-step process for proposing, approving, and executing transfers, particularly for restricted addresses, requiring a predefined number of approvals from designated approvers. It features three roles, `OWNER_ROLE` for managing restricted addresses and approvers, `APPROVER_ROLE` for voting on proposals, and `EXECUTOR_ROLE` for finalizing transfers, along with mechanisms to manage restricted address lists, update approver settings, and query pending proposals. This ensures transparent, secure, and decentralized token management, with flexibility for future upgrades.

ex1ICOv2 file

The Ex1ICO contract is an upgradeable, role-based smart contract designed to manage Initial Coin Offerings (ICOs) for the EX1 token, enabling secure token sales with support for USDC, USDT, ETH, and BTC payments. Leveraging OpenZeppelin's Initializable, ReentrancyGuardUpgradeable, AccessControlUpgradeable, and UUPSUpgradeable contracts, it facilitates the creation and management of ICO stages with customizable parameters like token price and duration, while integrating Chainlink price feeds for real-time ETH and BTC pricing. The contract supports on-chain purchases with USDC/USDT, off-chain ETH/BTC transactions recorded by authorized roles, and tracks tokens sold, funds raised, and buyer data, with purchase limits and administrative controls for setting parameters, ensuring a scalable, secure, and transparent token sale process.

ex1ICOVesting file

The ICOVesting contract is an upgradeable, role-based smart contract designed to manage the vesting and claiming of EX1 tokens purchased during an ICO, ensuring rewards are distributed over time on a linear basis. Built with OpenZeppelin libraries, it enables authorized users to create and update vesting schedules for specific ICO stages, defining start/end times, claim intervals, and slice periods for gradual token release. Token holders can claim their vested tokens based on these periodic schedules, with claimable amounts calculated linearly according to elapsed time, and reentrancy protection ensures security. Integrated with the Ex1ICO contract to verify user deposits, it supports role-based access (`OWNER` , `UPGRADER` , `VESTING_AUTHORISER`) for administrative tasks like updating schedules and interfaces, offering a secure and flexible vesting solution.

ex1Staking file

The Ex1Staking contract is an upgradeable smart contract designed to facilitate staking of EX1 tokens based on the amount purchased during an ICO and before the vesting period starts, incentivizing long-term holding through rewards. Built with OpenZeppelin libraries, it allows users to choose to stake their tokens, with staking parameters (percentage return, time period, and ICO stage) set by authorized roles. Rewards are calculated dynamically based on the staked amount and elapsed time, and users can claim them or unstake tokens at any point, with rewards adjusted accordingly. Integrated with the Ex1ICO and ICOVesting contracts to verify deposits and vesting schedules, it allows the staking of the user deposits

bofore the vesting. It supports role-based access (`OWNER` , `UPGRADER` , `STAKING_AUTHORISER`) for administrative updates, ensuring secure and transparent staking operations.

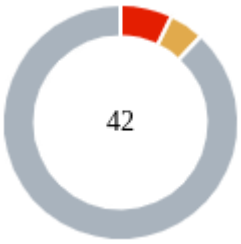
ex1EthICO file

The Ex1ICO contract is an upgradeable, role-based smart contract designed to facilitate the purchase of EX1 tokens using Ethereum (ETH) on the Ethereum blockchain, serving as a bridge for token sales that are ultimately recorded on the Binance Smart Chain (BNB Chain). Built with OpenZeppelin libraries, it enables the creation and management of ICO stages with customizable start/end times, token prices, and active status, while integrating a Chainlink price feed to calculate token prices in ETH. Users can buy tokens via ETH, with transactions recorded on Ethereum and mirrored on the BNB Chain ICO contract for token release on BNB Chain. The contract tracks total buyers, ETH raised, and user deposits per stage, enforces purchase limits, and supports role-based access (`OWNER` , `UPGRADER` , `ICO_AUTHORISER`) for administrative tasks like updating stage parameters and wallet addresses, ensuring a secure and efficient cross-chain token sale process.

ex1PrivateVesting file

The PrivateVesting contract is an upgradeable, role-based smart contract designed to manage token vesting schedules with role-based access control, specifically for addresses that do not participate in the ICO, ensuring controlled token distribution over time on a linear basis. Built with OpenZeppelin libraries, it allows authorized users to create, update, and revoke vesting schedules for beneficiaries, defining parameters like total amount, start/end times, claim intervals, cliff periods, and slice periods for gradual token release. Beneficiaries can claim vested tokens periodically after the cliff period, with claimable amounts calculated linearly based on elapsed time. The contract supports role-based access (`OWNER` , `UPGRADER` , `VESTING_CREATOR`) for administrative tasks, tracks vesting schedules and claim histories, and includes revocation options for revocable schedules, providing a secure and flexible vesting solution for private allocations.

Findings Breakdown



Critical	3
Medium	2
Minor / Informative	37

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	3	0	0	0
Medium	2	0	0	0
Minor / Informative	37	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IRAH	Inconsistent Restricted Address Handling	Unresolved
●	IEC	Incorrect ETH Calculations	Unresolved
●	ZD	Zero Division	Unresolved
●	IRL	Inconsistent Revocation Logic	Unresolved
●	UFF	Unstake Functionality Flaws	Unresolved
●	BC	Blacklists Addresses	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	DAC	Duplicate Access Checks	Unresolved
●	DVA	Duplicate Voting Allowed	Unresolved
●	IDU	Inconsistent Data Updates	Unresolved
●	IVU	Inefficient Variable Usage	Unresolved
●	IVS	Inefficient Vesting Setup	Unresolved

●	MVN	Misleading Variables Naming	Unresolved
●	MAPU	Missing Active Parameter Update	Unresolved
●	MAV	Missing Approver Validation	Unresolved
●	MCIC	Missing Claim Interval Check	Unresolved
●	MEVC	Missing EthICOStage Validation Checks	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MEC	Missing Existence Check	Unresolved
●	MRAV	Missing Required Approvers Validation	Unresolved
●	MTDC	Missing Token Decimal Check	Unresolved
●	MUV	Missing User Validation	Unresolved
●	MEV	Misspelled Enum Value	Unresolved
●	MU	Modifiers Usage	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	OSET	Overlapping Start End Times	Unresolved
●	POSD	Potential Oracle Stale Data	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	RPV	Redundant Parameter Validation	Unresolved
●	RRC	Redundant Role Checks	Unresolved

●	RSC	Redundant Staking Check	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	UTP	Unbounded Time Period	Unresolved
●	UPU	Unnecessary Parameter Usage	Unresolved
●	UEE	Unrefunded Excess ETH	Unresolved
●	UVS	Unused Variable Set	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

IRAH - Inconsistent Restricted Address Handling

Criticality	Critical
Location	ex1Token.sol#L212
Status	Unresolved

Description

The contract is designed to enforce multi-signature approval for token transfers from restricted addresses through the `isAddressRestricted` check in the `transfer` function, which triggers a proposal process if the sender is restricted. However, the `transferFrom` function, which handles token transfers on behalf of another address using an allowance, does not implement the same `isAddressRestricted` check. As a result, restricted addresses can bypass the multi-signature approval process by using `transferFrom` to move tokens, either by approving a third party or by being approved to transfer tokens. This inconsistency undermines the contract's security model, as it allows restricted addresses to transfer tokens without the intended governance oversight, potentially leading to unauthorized or untracked token movements.

```
    function transfer(address _to, uint256 _value) public virtual override
returns (bool) {
        if (isAddressRestricted(_msgSender()) == true) {
            _proposeTransfer(_msgSender(), _to, _value);
        } else {
            _transfer(_msgSender(), _to, _value);
        }
        return true;
    }
```

Recommendation

It is recommended to implement the `isAddressRestricted` check in the `transferFrom` function, mirroring the logic in the `transfer` function, to ensure that transfers involving restricted addresses (either as the from or msg.sender address) are subject to the multi-signature approval process. This would involve verifying whether the from address is restricted and, if so, initiating a transfer proposal instead of executing the transfer directly. Ensuring consistent handling of restricted addresses across both transfer

and transferFrom functions will strengthen the contract’s governance and security mechanisms.

IEC - Incorrect ETH Calculations

Criticality	Critical
Location	ex1EthICO.sol#L159
Status	Unresolved

Description

The contract is flawed due to incorrect calculations in the getTokenPriceInETH function, which determines the ETH equivalent for purchasing tokens, caused by improper decimal precision and scaling.

Example scenario:

Buying 100 tokens (100 *10^18) at a token price of 1 USD (1* 10^6) with an ETH price of 1500 USD (1.5 * 10^18).

USD Value: calculatePrice yields 100 USD (100 *10^6), as* `(100 * 10^18 * 1 * 10^6) / 10^18` = *100* 10^6`.

ETH Price Scaling: `getLatestETHPrice` function returns 150000000000 , scaled to 150000000000 * 10^10 in the `getTokenPriceInETH` function.

Incorrect Calculation: ethAmount = `(100 * 10^6 * 10^18) / (150000000000 * 10^10)` ≈ 66666.66666666667 wei.

Expected Calculation: 100 USD / 1500 USD/ETH ≈ 0.06666666666666667 ETH = 66666666666666666 wei.

This significant discrepancy results in users being overcharged in ETH, as the calculated amount is orders of magnitude lower than the correct value, disrupting token sale economics and potentially causing financial losses or deterring participation.

```
function getTokenPriceInETH(
        uint256 amount,
        uint256 _icoStageID
    ) public view returns (uint256 ethAmount) {
        uint256 usdPrice = calculatePrice(amount, _icoStageID);
        uint256 latestEthPrice = getLatestETHPrice() * (10 ** 10);
        uint256 _ethAmount = (usdPrice * (10 ** 18)) / latestEthPrice;
        return _ethAmount;
    }


    function calculatePrice(
        uint256 _amount,
        uint256 _icoStageID
    ) public view returns(uint256) {
         require (
            _amount < MaxTokenLimitPerTransaction,
            "ex1Presale: Max Limit Reached!"
        );
        require(
            icoStages[_icoStageID].isActive,
            "ex1Presale: Stage does not exist or is inactive!"
        );
        uint256 tokenValue = icoStages[_icoStageID].tokenPrice;
        uint256 usdValueUnscaled = (_amount * tokenValue)/(10 ** 18);
        return usdValueUnscaled;
    }

    function purchasedViaEth(
        uint256 _amount,
        uint256 _icoStageID
    ) external checkSaleStatus(_icoStageID) payable nonReentrant {
        require(
            block.timestamp >= icoStages[_icoStageID].startTime &&
block.timestamp <= icoStages[_icoStageID].endTime,
            "ex1Presale: Invalid Stage Paramaters"
        );
        uint256 ethAmount = getTokenPriceInETH(_amount, _icoStageID);
        require(
            msg.value >= ethAmount,
            "EthPayment: Insufficient Eths Value signed!"
        );
        ...
        }
```

Recommendation

It is recommended to revise the `getTokenPriceInETH` function to ensure accurate ETH calculations by correcting the decimal scaling and precision handling. The function should properly convert the USD value of tokens to ETH, accounting for all decimal places in the ETH price and token amounts. Implementing thorough unit tests to validate the function's output across various token quantities and ETH prices is crucial. Additionally, adding a refund mechanism in the purchasedViaEth function to return excess ETH sent by users will enhance fairness and user trust. These measures will align the contract's pricing logic with the intended token sale economics, preventing overcharges and ensuring accurate financial operations.

ZD - Zero Division

Criticality	Critical
Location	ex1ICOVesting.sol#L135,243 ex1PrivateVesting.sol#L329
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 `slicePeriod` can be set to zero.

```
  function updateClaimSchedule(
        uint256 _icoStageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _claimInterval,
        uint256 _slicePeriod
    ) external onlyRole(VESTING_AUTHORISER_ROLE) {
        ...
        );
        require(
            _slicePeriod >= 0 && _slicePeriod <= 60,
            "ex1Presale: Invalid Slice Period"
        );

        ...
    }

uint256 totalNumberOfSlices = (schedule.endTime - schedule.startTime) /
schedule.slicePeriod;
...
uint256 totalNumberOfSlices = (schedule.endTime - schedule.cliffPeriod) /
schedule.slicePeriod;
```

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.

IRL - Inconsistent Revocation Logic

Criticality	Medium
Location	ex1Token.sol#L246,273
Status	Unresolved

Description

The contract is not properly handling the revocation of approvals. Although the documentation suggests that `revokeApproval()` applies to transactions that have already been approved, the function can actually be called on transactions that have not been approved. Additionally, the `isRevoked` variable, which appears to be intended to track the revocation status, is never updated within the contract. This oversight means that even if a transaction gathers enough revocation votes to meet the threshold and is marked as revoked, the same transaction can still be approved if the required number of approval votes is subsequently reached. This discrepancy can lead to inconsistent logic and unexpected behavior, undermining the intended security and governance model.

```
 function approveTransfer(uint256 _txIndex)
        public
        onlyRole(APPROVER_ROLE)
        txExists(_txIndex)
        notExecuted(_txIndex)
        notVoted(_txIndex)
    {
        require(isApprover[_msgSender()] == true, "Approver Status
Held!");
        require(isRevoked[_txIndex] == false, "Transaction status
Revoked!");
        ...

        emit TransferApproved(_txIndex, _msgSender());

        if (transactions[_txIndex].approvalCount >= required) {
            _executeTransfer(_txIndex);
        }
    }

    function revokeApproval(uint256 _txIndex)
        public
        onlyRole(APPROVER_ROLE)
        txExists(_txIndex)
        notExecuted(_txIndex)
    {
        require(isApprover[_msgSender()] == true, "Approver Status
Held!");
        require(isRevoked[_txIndex] == false, "Transaction status
Revoked!");
        require(hasVoted[_txIndex][_msgSender()] == false, "Approver
already Voted");
        require(transactions[_txIndex].executed == false, "Transaction
already Executed");

        voteNature[_txIndex][_msgSender()] = 2;
        transactions[_txIndex].revokeCount += 1;

        if (transactions[_txIndex].revokeCount > required) {
            transactions[_txIndex].approvalStatus =
ApprovalStatus.revoked;
        }

        emit ApprovalRevoked(_txIndex, _msgSender());
    }
```

Recommendation

It is recommended that the team reconsider the `revokeApproval()` functionality and implement the logic correctly to align with the documentation. This includes ensuring that revocation can only occur on previously approved transactions and updating `isRevoked` or another similar state variable accordingly. The contract should enforce a consistent and clearly defined approval/revocation workflow to maintain the integrity of its access control and decision-making process.

UFF - Unstake Functionality Flaws

Criticality	Medium
Location	ex1Staking.sol#L314
Status	Unresolved

Description

The contract is designed to allow users to unstake their tokens, but the unstake function does not automatically calculate and distribute the staking rewards that users are entitled to at the time of unstaking, requiring a separate call to `claimStakingRewards`. Additionally, the function lacks restrictions on multiple calls, allowing users to repeatedly invoke it, allowing for updates to latest `unstakeTimestamp` and potentially spamming the system and increasing gas costs or network load without meaningful effect. Furthermore, if a user re-stakes tokens for the same ICO stage after unstaking but before claiming rewards, the unstaked flag is overwritten, and the previous unclaimed rewards are effectively lost due to the reset of staking-related state variables. This creates a risk of reward loss and undermines the reliability of the staking mechanism, as users may unintentionally forfeit rewards or exploit the system by repeatedly unstaking.

```
    function unstake(
        uint256 _icoStageID
    ) external returns(bool) {
        require(
            isStaked[_icoStageID][_msgSender()],
            "ex1Staking: Not Staked Yet!"
        );

        unstaked[_icoStageID][_msgSender()] = true;
        unstakeTimestamp[_icoStageID][_msgSender()] = block.timestamp;

        formerTotalStakedPerICO[_icoStageID][_msgSender()] =
totalStakedPerICO[_icoStageID][_msgSender()];
        totalStakedPerICO[_icoStageID][_msgSender()] = 0;

        return true;
    }
```

Recommendation

It is recommended to enhance the `unstake` function by automatically calculating and distributing any outstanding staking rewards at the time of unstaking, ensuring users receive their entitled rewards without requiring a separate transaction. Additionally, implement a check to prevent multiple unstake calls for the same ICO stage by verifying the unstaked status before processing. To address the issue of re-staking, consider disallowing re-staking for the same ICO stage after unstaking unless all rewards have been claimed, or maintain a separate record of unclaimed rewards to prevent loss. Emitting events for unstake actions and reward distributions can further improve transparency and auditability, ensuring a robust and user-friendly staking system.

BC - Blacklists Addresses

Criticality	Minor / Informative
Location	ex1Token.sol#L165,212
Status	Unresolved

Description

The `OWNER_ROLE` has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `addRestrictedAddress` function.

```
   function addRestrictedAddress(address _address) external
onlyRole(OWNER_ROLE) {
        restrictedAddresses.add(_address);
    }

    function transfer(address _to, uint256 _value) public virtual
override returns (bool) {
        if (isAddressRestricted(_msgSender()) == true) {
            _proposeTransfer(_msgSender(), _to, _value);
        } else {
            _transfer(_msgSender(), _to, _value);
        }
        return true;
    }
```

Recommendation

The team should carefully manage the private keys of the `OWNER_ROLE'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.

CR - Code Repetition

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L345,396
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
function buyWithUSDC(
        uint256 _amount,
        uint256 _icoStageID,
        IERC20 _token
    ) external checkSaleStatus(_icoStageID) returns(bool) {
        ...

        return true;
    }

    /**
        @dev Function to buy tokens with USDT
    */
    function buyWithUSDT(
        uint256 _amount,
        uint256 _icoStageID,
        IERC20 _token
    ) external checkSaleStatus(_icoStageID) returns(bool) {
        ...

        return true;
    }

   function purchasedViaEth(
        uint256 _amount,
        uint256 _ethRecieved,
        uint256 _icoStageID,
        address _recipient
    ) external checkSaleStatus(_icoStageID) onlyRole(TXN_RECORDER_ROLE) {

        ...
    }


    function purchasedViaBTC(
        uint256 _amount,
        uint256 _btcRecieved,
        uint256 _icoStageID,
        address _recipient
    ) external checkSaleStatus(_icoStageID) onlyRole(TXN_RECORDER_ROLE) {

        ....
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	ex1ICOv2.sol ex1Staking.sol ex1PrivateVesting.sol ex1ICOVesting.sol ex1EthICO.sol
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 contract is heavily reliant on centralized role authorities, such as `OWNER_ROLE`, `UPGRADER_ROLE`, and `ICO_AUTHORISER_ROLE`, to configure, modify, and manage critical contract parameters. This centralization introduces a potential risk. If the individuals or entities holding these roles make an error, act maliciously, or fail to act in a timely manner, the entire contract and its associated processes can be compromised. Without proper checks and balances, the system's integrity depends on the correct and honest execution of these centralized roles, making it vulnerable to human error or abuse of power.

```
bytes32 public constant OWNER_ROLE = keccak256("OWNER_ROLE");
bytes32 public constant UPGRADER_ROLE = keccak256("UPGRADER_ROLE");
bytes32 public constant ICO_AUTHORISER_ROLE =
keccak256("ICO_AUTHORISER_ROLE");
```

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.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L330 ex1ICOVesting.sol#L244
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 tokenPerSlice = schedule.totalAmount / totalNumberOfSlices;
...
uint256 tokenPerSlice = totalDeposits / totalNumberOfSlices;
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

DAC - Duplicate Access Checks

Criticality	Minor / Informative
Location	ex1Token.sol#L248,273,298
Status	Unresolved

Description

The contract includes `require` checks that are already enforced by the function's modifiers. This redundancy leads to unnecessary complexity and potential confusion, as the same conditions are validated multiple times. While modifiers already confirm these requirements before the function body executes, additional require statements within the function repeat these checks, creating redundancy without adding meaningful security benefits.

```
  function approveTransfer(uint256 _txIndex)
        public
        onlyRole(APPROVER_ROLE)
        txExists(_txIndex)
        notExecuted(_txIndex)
        notVoted(_txIndex)
    {
        require(isApprover[_msgSender()] == true, "Approver Status
Held!");
        ...
        require(hasVoted[_txIndex][_msgSender()] == false, "Approver
already Voted");

    ...
    function revokeApproval(uint256 _txIndex)
        public
        onlyRole(APPROVER_ROLE)
        txExists(_txIndex)
        notExecuted(_txIndex)
    {
        require(isApprover[_msgSender()] == true, "Approver Status
Held!");
        require(isRevoked[_txIndex] == false, "Transaction status
Revoked!");
        ...

    function _executeTransfer(uint256 _txIndex)
        internal
        txExists(_txIndex)
        notExecuted(_txIndex)
    {
        require(transactions[_txIndex].approvalCount >= required, "not
enough approvals");

        Transaction storage transaction = transactions[_txIndex];
        require(!transaction.executed, "transfer already executed");
        ...
```

Recommendation

It is recommended to review and reconsider the access control logic and the modifiers applied to these functions. By removing duplicate require statements that are already covered by modifiers, the codebase can become more streamlined, easier to read, and less prone to maintenance errors.

DVA - Duplicate Voting Allowed

Criticality	Minor / Informative
Location	ex1Token.sol#L273
Status	Unresolved

Description

The `revokeApproval` function does not set the `hasVoted` mapping to `true` when an approver casts a revocation vote. This omission means that the system does not properly record that the approver has participated in the voting process for a given transaction. As a result, the same approver may be able to repeatedly call the `revokeApproval` function, skewing the revocation count and potentially leading to unintended outcomes.

```
    function revokeApproval(uint256 _txIndex)
        public
        onlyRole(APPROVER_ROLE)
        txExists(_txIndex)
        notExecuted(_txIndex)
    {
        require(isApprover[_msgSender()] == true, "Approver Status
Held!");
        require(isRevoked[_txIndex] == false, "Transaction status
Revoked!");
        require(hasVoted[_txIndex][_msgSender()] == false, "Approver
already Voted");
        require(transactions[_txIndex].executed == false, "Transaction
already Executed");

        voteNature[_txIndex][_msgSender()] = 2;
        transactions[_txIndex].revokeCount += 1;

        if (transactions[_txIndex].revokeCount > required) {
            transactions[_txIndex].approvalStatus =
ApprovalStatus.revoked;
        }

        emit ApprovalRevoked(_txIndex, _msgSender());
    }
```

Recommendation

It is recommended to update the `revokeApproval` function so that once an approver votes to revoke, the corresponding `hasVoted` entry is set to `true`. This change will ensure that each approver can only participate once per transaction, maintaining the integrity of the voting process and preventing duplicate voting.

IDU - Inconsistent Data Updates

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L235 ex1EthICO.sol#L144 ex1ICOVesting.sol#L173 ex1Staking.sol#L111
Status	Unresolved

Description

The contract is managing multiple functionalities, presale, vesting, and staking, based on a shared `_icoStageID`. However, updates to variables such as the stage parameters (e.g. in ICO) occur independently within each contract, without a mechanism to propagate these changes across the related functionalities. For example, if the ICO stage parameters are modified in one contract, other contracts that depend on these parameters may not reflect the updated values. This disconnect can lead to inconsistencies, as calculations or conditions in one contract may no longer align with the updated data in another.

```
function updateICOStage(
        uint256 _stageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        bool active
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        ...
    }
```

```
   function updateICOStage(
        uint256 _stageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
    ...
```

```
    function claimTokens(
         uint256 _icoStageID
     ) external nonReentrant {
        ...
         uint256 deposits = icoInterface.UserDepositsPerICOStage(
_icoStageID, _msgSender());
         require(deposits > 0, "ex1Presale: No Tokens to Claim!");
         ...
```

```
    function stake(
         uint256 _icoStageID
     ) external returns(bool) {
         uint256 deposits =
icoInterface.UserDepositsPerICOStage(_icoStageID, _msgSender());
         ( , uint256 startTime, , , ) =
vestingInterface.claimSchedules(_icoStageID);
         ...
         }
```

Recommendation

It is recommended to implement a decentralized mechanism or shared interface that ensures changes to ICO stage parameters are consistently applied across all related contracts. This could involve storing the parameters in a single source of truth or using events and callbacks to notify dependent contracts of updates. By aligning data and logic across all functionalities, the system can prevent inconsistencies, reduce maintenance overhead, and improve reliability.

IVU - Inefficient Variable Usage

Criticality	Minor / Informative
Location	ex1ICOVesting.sol#L325
Status	Unresolved

Description

The contract uses an extra variable to hold the calculated balance before returning it. This adds unnecessary complexity, as the calculation could be done inline and returned directly. By doing so, the code can be made more concise and potentially slightly more gas-efficient, without altering the logic or the outcome.

```
if(block.timestamp > schedule.endTime) {
        uint256 balance = schedule.totalAmount -
schedule.releasedAmount;
        return balance;
    }
```

Recommendation

It is recommended to eliminate the extra variable and return the result of the calculation directly. This will simplify the code, reduce storage overhead, and improve clarity.

IVS - Inefficient Vesting Setup

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L144
Status	Unresolved

Description

The current approach requires a central role (VESTING_CREATOR_ROLE) to manually set the vesting schedules for all participating users. This centralized process is inefficient and gas-intensive, especially when the number of beneficiaries increases. Additionally, handling multiple user entries in this manner increases the risk of errors or incorrect data entries, potentially causing mismanagement of vesting schedules and related funds.

```
function setVestingSchedule(
        address _beneficiary,
        uint256 _totalAmount,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _claimInterval,
        uint256 _cliffPeriod,
        uint256 _slicePeriod,
        bool _isRevocable
    ) external onlyRole(VESTING_CREATOR_ROLE) {
        ...
        latestVestingScheduleID ++;

        vestingSchedules[latestVestingScheduleID] = VestingSchedule({
            beneficiary: _beneficiary,
            vestingScheduleID: latestVestingScheduleID,
            totalAmount: _totalAmount,
            startTime: _startTime,
            endTime: _endTime,
            claimInterval: _claimInterval,
            cliffPeriod: _cliffPeriod,
            slicePeriod: _slicePeriod,
            releasedAmount: 0,
            isRevocable: _isRevocable,
            isRevoked: false
        });

        ...
    }
```

Recommendation

It is recommended to consider a more decentralized approach to vesting schedule creation and validation. By allowing users or their authorized agents to initiate their own vesting schedules—while still adhering to contract-defined rules and checks—this would reduce the gas cost per action and minimize the administrative overhead for a single role. Implementing self-service or automated validation mechanisms can enhance efficiency and reduce the likelihood of incorrect data entries.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L482,535
Status	Unresolved

Description

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

The `totalTokensSold` variable is incremented regardless of whether tokens are actually transferred to the recipient. In scenarios where tokens remain unreleased and are only added to a user's deposit balance, the `totalTokensSold` counter still increases. This behavior can cause confusion, as the variable's name implies that it tracks actual token sales, yet it is updated even when no tokens are transferred.

```
        totalETHRaised += _ethRecieved;

         BoughtWithEth[_recipient] += _ethRecieved;
         ...

         totalBTCRaised += _btcRecieved;

         BoughtWithBTC[_recipient] += _btcRecieved;
```

Recommendation

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

It is recommended to adjust the logic so that the variables only increases when tokens are truly sold and transferred to the recipient. Alternatively, the variable could be renamed to

more accurately reflect what it is measuring, such as “totalTokensAllocated” to ensure consistency between the variable’s name and its actual behavior.

MAPU - Missing Active Parameter Update

Criticality	Minor / Informative
Location	ex1EthICO.sol#L144
Status	Unresolved

Description

The `updateICOStage` function does not include the `_isActive` parameter, which prevents the contract from updating the stage's active status. While the `createICOStage` function allows setting the `_isActive` state, the corresponding update function lacks the ability to modify this field. As a result, the contract cannot reactivate or deactivate a previously created ICO stage.

```
function createICOStage(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        uint256 _icoStageID,
        bool _isActive
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        icoStages[_icoStageID] = ICOStage({
            startTime: _startTime,
            endTime: _endTime,
            stageID: _icoStageID,
            tokenPrice: _tokenPriceUSD,
            isActive: _isActive
        });

        stageIDs.push(_icoStageID);

        emit ICOStageCreated(
            _icoStageID,
            _startTime,
            _endTime,
            _tokenPriceUSD,
            _isActive
        );
    }


    function updateICOStage(
        uint256 _stageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        icoStages[_stageID].startTime = _startTime;
        icoStages[_stageID].endTime = _endTime;
        icoStages[_stageID].tokenPrice = _tokenPriceUSD;
        emit ICOStageUpdated(_stageID, _startTime, _endTime,
_tokenPriceUSD);
    }
```

Recommendation

It is recommended to add the `_isActive` parameter to the `updateICOStage` function. By including this field, the function can maintain consistency with the `createICOStage` logic and provide full control over all attributes of an ICO stage, including its active status.

MAV - Missing Approver Validation

Criticality	Minor / Informative
Location	ex1Token.sol#L138
Status	Unresolved

Description

The contract lacks a validation mechanism in the `updateApprovers` function to confirm that addresses in the `_approver` array are listed in the approvers array before their `isApprover` status is updated. This could permit the DEFAULT_ADMIN_ROLE to alter the status of non-existent or unauthorized addresses, potentially leading to inconsistencies in the multi-signature approval process and affecting the reliability of the contract's governance structure.

```
    function updateApprovers(address[] memory _approver, bool[] memory
status) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_approver.length == status.length, "Length Mismatched!");
        for(uint i = 0; i< _approver.length; i++) {
            isApprover[_approver[i]] = status[i];
        }
    }
```

Recommendation

It is recommended to add a validation check in the `updateApprovers` function to ensure each address in the `_approver` array is a valid member of the approvers array before modifying its `isApprover` status. Incorporating an event emission for status changes can further improve transparency and facilitate auditing of governance actions.

MCIC - Missing Claim Interval Check

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L255
Status	Unresolved

Description

The contract is missing a validation check in the `updateVesting` function to prevent the `_claimInterval` from being set to zero. While the `setVestingSchedule` function includes a condition that ensures `_claimInterval` is greater than zero, this safeguard is not present in `updateVesting`. As a result, it is possible to set an invalid `_claimInterval` in an updated schedule, potentially causing unexpected or incorrect behavior.

```
function setVestingSchedule(
        address _beneficiary,
        uint256 _totalAmount,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _claimInterval,
        uint256 _cliffPeriod,
        uint256 _slicePeriod,
        bool _isRevocable
    ) external onlyRole(VESTING_CREATOR_ROLE) {
        ...
        require(
            _claimInterval <= (_endTime - _cliffPeriod)
            && _claimInterval > 0,
            "Private: Invalid Claim Interval"
        );
        ...
    }


    function updateVesting(
        uint256 _vestingScheduleID,
        address _beneficiary,
        uint256 _totalAmount,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _claimInterval,
        uint256 _cliffPeriod,
        uint256 _slicePeriod
    ) external onlyValidSchedule(_vestingScheduleID)
onlyRole(VESTING_CREATOR_ROLE){
        ...
        require(
            _claimInterval <= (_endTime - _startTime),
            "PrivateVesting: Invalid Cliff Period"
        );
        ...
        }
```

Recommendation

It is recommended to add a validation step in the `updateVesting` function to ensure that `_claimInterval` is greater than zero. By including this check, the contract will maintain consistent standards for schedule intervals and prevent unintended configurations that could disrupt the vesting process.

MEVC - Missing EthICOStage Validation Checks

Criticality	Minor / Informative
Location	ex1EthICO.sol#L111,144
Status	Unresolved

Description

The contract is missing several key verification checks in both the `createICOStage` and `updateICOStage` functions. Specifically, `createICOStage` does not ensure that the same `_icoStageID` is not reused, the `endTime` is not validated to be greater than `startTime`, and `startTime` is not checked to ensure it is in the future (greater than the current block timestamp). Similarly, the `updateICOStage` function lacks these same verification steps. These missing checks leave the contract vulnerable to invalid input data, which could lead to incorrect stage configurations or unexpected behavior. Additionally, these omissions mean the functions do not align with the established verification standards used in the `Ex1ICO` contract on the BNB Chain.

```
function createICOStage(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        uint256 _icoStageID,
        bool _isActive
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        icoStages[_icoStageID] = ICOStage({
            startTime: _startTime,
            endTime: _endTime,
            stageID: _icoStageID,
            tokenPrice: _tokenPriceUSD,
            isActive: _isActive
        });

        stageIDs.push(_icoStageID);

        ...
    }


    function updateICOStage(
        uint256 _stageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        icoStages[_stageID].startTime = _startTime;
        icoStages[_stageID].endTime = _endTime;
        icoStages[_stageID].tokenPrice = _tokenPriceUSD;
        emit ICOStageUpdated(_stageID, _startTime, _endTime,
_tokenPriceUSD);
    }
```

Recommendation

It is recommended to add validation checks to both functions. The `createICOStage` function should verify that `_icoStageID` is unique, `endTime` is greater than `startTime`, and `startTime` is greater than the current block timestamp. Likewise, the `updateICOStage` function should include these same checks to maintain consistency and ensure that all ICO stages are properly configured. Including these validations will reduce the risk of configuration errors and align the logic with best practices.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	ex1Token.sol#L138 ex1ICOv2.sol#L571 ex1PrivateVesting.sol#L225
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 updateApprovers(address[] memory _approver, bool[] memory
status) external onlyRole(DEFAULT_ADMIN_ROLE) {
    ...
    }

    function addApprover(address[] memory _approver) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    ...
    }
```

```
function setReceiverWallet(address _wallets) external
onlyRole(OWNER_ROLE) {
        require(
            _wallets != address(0),
            "ex1Presale: Invalid Wallet Address!"
        );
        recievingWallet = _wallets;
    }

    function setTokenReleasable() external onlyRole(OWNER_ROLE) {
        isTokenReleasable = !isTokenReleasable;
    }

    function setIAggregatorInterfaceETH(IAggregator _aggregator) external
onlyRole(OWNER_ROLE) {
        require(
            address(_aggregator) != address(0),
            "ex1Presale: Invalid Aggregator Address!"
        );
        aggregatorInterfaceETH = _aggregator;
    }
```

```
       function updateVesting(
        uint256 _vestingScheduleID,
        address _beneficiary,
        uint256 _totalAmount,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _claimInterval,
        uint256 _cliffPeriod,
        uint256 _slicePeriod
    ) external onlyValidSchedule(_vestingScheduleID)
onlyRole(VESTING_CREATOR_ROLE){
        ...
        }
```

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.

MEC - Missing Existence Check

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L135
Status	Unresolved

Description

The `updateClaimSchedule` function does not validate whether a claim schedule already exists for the provided `_icoStageID` before attempting to update it. Without this check, it is possible for the contract to update a non-existent claim schedule, potentially resulting in undefined or unexpected behavior.

```
function updateClaimSchedule(
        uint256 _icoStageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _claimInterval,
        uint256 _slicePeriod
    ) external onlyRole(VESTING_AUTHORISER_ROLE) {
        if(block.timestamp > claimSchedules[_icoStageID].startTime) {
            require(
                _startTime == claimSchedules[_icoStageID].startTime,
                "ex1Presale: Claim Schedule Already Started!"
            );
        }
       ...
    }
```

Recommendation

It is recommended to add a validation step that ensures a claim schedule exists for the given `_icoStageID` before proceeding with updates. By confirming that the relevant claim schedule is already defined, the contract can prevent accidental or unintended updates to uninitialized claim schedules, improving both security and reliability.

MRAV - Missing Required Approvers Validation

Criticality	Minor / Informative
Location	ex1Token.sol#L117,352
Status	Unresolved

Description

The contract is missing a validation check in the `updateRequiredApprovers` function that was present during initialization. The original logic ensures that the `_required` parameter falls within a valid range, specifically that `_required > 0` and `_required <= _approver.length`. Without this check, it becomes possible to set an invalid number of required approvers, potentially leading to incorrect or unintended contract behavior.

```
        require(_required > 0 && _required <= _approver.length, "invalid
required number");
    ...

    function updateRequiredApprovers(uint256 _required) external
onlyRole(OWNER_ROLE) {
        required = _required;
    }
```

Recommendation

It is recommended to implement the same validation check in `updateRequiredApprovers` as was used during initialization. Adding this check will ensure consistency and prevent invalid configurations, thereby improving the contract's reliability and reducing the likelihood of errors or misuse.

MTDC - Missing Token Decimal Check

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L599
Status	Unresolved

Description

The contract does not validate that the newly assigned token addresses have matching decimals. Since the functions `setUSDTAddress` and `setUSDCAddress` allow changing the token address without verifying the decimals, any subsequent calculations that depend on consistent token units may produce incorrect results. This issue can cause unintended behavior and may lead to problems if the tokens differ in decimal precision.

```
   function setUSDTAddress(IERC20 _USDTTokenAddress) external
onlyRole(OWNER_ROLE) {
        require(
            address(_USDTTokenAddress) != address(0),
            "EX1Presale: Invalid USDT Address!"
        );
        USDTAddress = _USDTTokenAddress;
    }

    function setUSDCAddress(IERC20 _USDCTokenAddress) external
onlyRole(OWNER_ROLE) {
        require(
            address(_USDCTokenAddress) != address(0),
            "EX1Presale: Invalid USDC Address!"
        );
        USDCAddress = _USDCTokenAddress;
    }
```

Recommendation

It is recommended to include a check to ensure that the newly set token address has the same decimal value as the previous token. This can be done by verifying the `decimals()` function of the ERC20 token contract before assigning the new address. Adding this validation will help maintain consistent unit calculations and prevent errors caused by differing token decimals.

MUV - Missing User Validation

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L388,439
Status	Unresolved

Description

The contract relies on the user providing approval for the transfer of USDT and USDC tokens. If the user has not already approved the required amount, the `safeTransferFrom` call will fail. However, the current implementation does not include a check to verify that the required approval is in place before attempting the transfer. As a result, the user may encounter a transaction failure without a clear error message indicating the need for token approval, which can lead to confusion and a poor user experience.

```
IERC20(_token).safeTransferFrom(_msgSender(), recievingWallet,
usdValue);
```

Recommendation

It is recommended to implement a validation step that ensures the user has approved the required amount of USDT and USDC tokens prior to executing the transfer. If no approval is detected, the contract should provide a specific error message that clearly explains the issue. This enhancement will help users understand why the transaction failed and what action they need to take, improving the overall usability and transparency of the contract.

MEV - Misspelled Enum Value

Criticality	Minor / Informative
Location	ex1Token.sol#L24
Status	Unresolved

Description

The `ApprovalStatus` enum includes a value labeled as `exectued`, which appears to be a misspelling of the word "executed." This typographical error can lead to confusion, inconsistencies, and potential issues during development, as developers may not immediately recognize that `exectued` refers to the executed state.

```
    enum ApprovalStatus {
        pending,
        revoked,
        exectued
    }
```

Recommendation

It is recommended to correct the misspelling of `exectued` to `executed`. This change will ensure clarity and maintain standard naming conventions, reducing the likelihood of errors or misunderstandings when referencing the enum values.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	ex1Token.sol#L255,281 ex1ICOv2.sol#L584 ex1PrivateVesting.sol#L154
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(transactions[_txIndex].executed == false, "Transaction already
Executed");
require(hasVoted[_txIndex][_msgSender()] == false, "Approver already
Voted");
```

```
require(
     block.timestamp >= icoStages[_icoStageID].startTime &&
block.timestamp <= icoStages[_icoStageID].endTime,
     "ex1Presale: Invalid Stage Paramaters"
 );
...
require(
    address(_aggregator) != address(0),
    "ex1Presale: Invalid Aggregator Address!"
);
```

```
require(
    _startTime > 0
    && _endTime > 0
    && _totalAmount > 0
    && _beneficiary != address(0),
    "PrivateVesting: Value cannot be 0 or 0 address!"
);
require(
    _endTime > _startTime
    && _startTime > block.timestamp,
    "Private: Invalid Time Schedule"
);
require(
    _cliffPeriod > _startTime
    && _cliffPeriod < _endTime,
    "PrivateVesting: Invalid Cliff Period"
);
...
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	ex1EthICO.sol#L159
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
   function getLatestETHPrice() public view returns (uint256) {
        (, int256 price, , , ) =
aggregatorInterfaceETH.latestRoundData();
        return uint256(price);
    }


    function getTokenPriceInETH(
        uint256 amount,
        uint256 _icoStageID
    ) public view returns (uint256 ethAmount) {
        uint256 usdPrice = calculatePrice(amount, _icoStageID);
        uint256 latestEthPrice = getLatestETHPrice() * (10 ** 10);
        uint256 _ethAmount = (usdPrice * (10 ** 18)) / latestEthPrice;
        return _ethAmount;
    }
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

OSET - Overlapping Start End Times

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L235
Status	Unresolved

Description

The `updateICOStage` function currently verifies that the new start time does not overlap with the end time of the previous stage and that the new end time does not overlap with the start time of the next stage. However, it does not ensure that the new start time does not precede the start time of the previous stage, nor does it confirm that the new end time does not occur before the end time of the previous stage. These missing checks could lead to inconsistencies in the timeline of ICO stages, potentially causing confusion or unintended behavior.

```
function updateICOStage(
        uint256 _stageID,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        bool active
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        require(_stageID <= latestICOStageID, "ex1Presale: Stage does not
exist");
        require(
            (_startTime < _endTime) &&
            _endTime > block.timestamp,
            "ex1Presale: Invalid time range!"
        );
        uint256 prevID = _stageID - 1;
        uint256 nextID = _stageID + 1;
        if(_stageID > 1) {
            require(
                _startTime > icoStages[prevID].endTime,
                "ICO: Start Time Overlapping with previous ICO end Time!"
            );
        }
        if(_stageID < latestICOStageID) {
            require(
                _endTime < icoStages[nextID].startTime,
                "ICO: End Time Overlapping with next ICO startTime!"
            );
        }
        ...
    }
```

Recommendation

It is recommended to include checks ensuring that the new start time is not before the start time of the previous stage, and that the new end time is not before the end time of the previous stage. This will help maintain a logical and consistent timeline of ICO stages, reducing the risk of overlapping or misaligned stage intervals.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	ex1EthICO.sol#L159
Status	Unresolved

Description

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

```
    function getLatestETHPrice() public view returns (uint256) {
        (, int256 price, , , ) =
aggregatorInterfaceETH.latestRoundData();
        return uint256(price);
    }


    function getTokenPriceInETH(
        uint256 amount,
        uint256 _icoStageID
    ) public view returns (uint256 ethAmount) {
        uint256 usdPrice = calculatePrice(amount, _icoStageID);
        uint256 latestEthPrice = getLatestETHPrice() * (10 ** 10);
        uint256 _ethAmount = (usdPrice * (10 ** 18)) / latestEthPrice;
        return _ethAmount;
    }
```

Recommendation

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

update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

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

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

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	ex1EthICO.sol#L224
Status	Unresolved

Description

The contract sends funds to a `marketingWallet` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
(bool success, ) = payable(receivingWallet).call{value: msg.value}("");
require(
    success,
    "Ex1 ETH: Transfer of ETH failed"
);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

RPV - Redundant Parameter Validation

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L114,134,345 ex1PrivateVesting.sol#L154,243
Status	Unresolved

Description

The contract is performing checks that overlap or contradict each other, resulting in redundancy. For example, the `_endTime` parameter is verified both to be greater than zero and to be greater than the current block timestamp. These overlapping conditions not only make the code more complex but also increase the potential for confusion, as certain checks appear to validate the same requirement multiple times.

```
    function createICOStage(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        bool _isActive
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        ...
        require (
            (_startTime < _endTime) && (_startTime > 0 || _endTime > 0),
            "ex1Presale: Invalid Schedule or Parameters!"
        );
        require (
            _startTime > block.timestamp,
            "ex1Presale: Invalid Start Time!"
        );
        require (
            _endTime > block.timestamp,
            "ex1Presale: Invalid End Time!"
        );


...
    modifier checkSaleStatus(uint256 _icoStageID) {
        require(
            block.timestamp >= icoStages[_icoStageID].startTime &&
            block.timestamp <= icoStages[_icoStageID].endTime &&
            icoStages[_icoStageID].isActive,
            "ex1Presale: Sale Not Active!"
        );
        _;
    }

    function buyWithUSDC(
        uint256 _amount,
        uint256 _icoStageID,
        IERC20 _token
    ) external checkSaleStatus(_icoStageID) returns(bool) {
        ...
        require(
            block.timestamp >= icoStages[_icoStageID].startTime &&
block.timestamp <= icoStages[_icoStageID].endTime,
            "ex1Presale: Invalid Stage Paramaters"
        );
```

```
require(
     _startTime > 0
     && _endTime > 0
     && _totalAmount > 0
     && _beneficiary != address(0),
     "PrivateVesting: Value cannot be 0 or 0 address!"
 );
 require(
     _endTime > _startTime
     && _startTime > block.timestamp,
     "Private: Invalid Time Schedule"
 );
 require(
     _cliffPeriod > _startTime
     && _cliffPeriod < _endTime,
     "PrivateVesting: Invalid Cliff Period"
 );
```

Recommendation

It is recommended that the contract's parameter validation logic be reviewed and streamlined. By eliminating redundant checks, the code can be made clearer and more efficient. Only the necessary conditions should be retained, ensuring that the contract maintains a concise and easily understandable validation process.

RRC - Redundant Role Checks

Criticality	Minor / Informative
Location	ex1Token.sol#L315
Status	Unresolved

Description

The contract is introducing redundancy by requiring both the `DEFAULT_ADMIN_ROLE` and the `EXECUTOR_ROLE` for executing certain functions. The `DEFAULT_ADMIN_ROLE` already has the authority to set approvals and manage roles, making the additional check for the `EXECUTOR_ROLE` superfluous. This overlap unnecessarily complicates the access control structure and can cause confusion, potentially increasing the risk of implementation or operational errors.

```
    function executeTransfer(uint256 _txIndex)
        external
        txExists(_txIndex)
        notExecuted(_txIndex)
        onlyRole(DEFAULT_ADMIN_ROLE)
        onlyRole(EXECUTOR_ROLE)
    {
        _executeTransfer(_txIndex);
    }
```

Recommendation

It is recommended that the team re-evaluate the necessity of having both roles in this context. Removing the redundant role check and relying on the `DEFAULT_ADMIN_ROLE` alone can streamline the access control logic, reduce complexity, and make the contract easier to understand and maintain.

RSC - Redundant Staking Check

Criticality	Minor / Informative
Location	ex1Staking.sol#L141
Status	Unresolved

Description

The `calculateStakeReward` function contains a redundant `require` statement verifying that the caller is already staked. This condition is unnecessary since the calling function, `claimStakingRewards`, has already performed the same check. Repeating the check introduces unnecessary code complexity and does not add meaningful security or logic benefits.

```
    function claimStakingRewards(
        uint256 _icoStageID
    ) external nonReentrant {
        require(
            isStaked[_icoStageID][_msgSender()],
            "ex1Staking: Not Staked Yet!"
        );
        ...
    }

    function calculateStakeReward(
        uint256 _icoStageID,
        address _caller
    ) internal returns(uint256) {
        require(
            isStaked[_icoStageID][_caller],
            "ex1Staking: Not Staked Yet!"
        );
```

Recommendation

It is recommended to remove the redundant `require` statement in `calculateStakeReward`. This will streamline the code and reduce duplication, improving readability and maintainability without compromising functionality.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L370,421,466,519 ex1ICOVesting.sol#L209 ex1Staking.sol#L153
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.

```
ERC20(ex1Token).safeTransfer(msg.sender, _amount);
...
bool success = IERC20(ex1Token).transfer(_recipient, _amount);
require(
    success,
    "ex1Presale: Token Transfer Failed!"
);
```

```
ex1Token.safeTransfer(_msgSender(), claimableAmount);
```

```
bool success = IERC20(ex1Token).transfer(_msgSender(), reward);
```

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.

UTP - Unbounded Time Period

Criticality	Minor / Informative
Location	ex1Staking.sol#L77
Status	Unresolved

Description

The contract is missing a validation check to ensure that the `timePeriodInSeconds` parameter remains within the intended bounds. Specifically, when creating staking reward parameters, the value of `timePeriodInSeconds` is not verified to fit within the logical window between the current block timestamp and the calculated staking end time (i.e., the start of vesting for the ICO stage). Without this validation, it is possible to set a duration that either exceeds the available timeframe or overlaps inappropriately with the vesting schedule. This can lead to misconfigured staking parameters, inaccurate reward distributions, or a broken user experience if staking periods overlap with vesting start times or result in staking windows that are practically unreachable.

```
function createStakingRewardsParamaters(
        uint256 _percentageReturn,
        uint256 _timePeriodInSeconds,
        uint256 _icoStageID
    ) external onlyRole(STAKING_AUTHORISER_ROLE) {
        ...
        require(
            _timePeriodInSeconds > 0,
            "ex1Staking: Invalid Time Period!"
        );

        ...
        stakingParameters[_icoStageID] = StakingParamter({
            percentageReturn: _percentageReturn,
            timePeriodInSeconds: _timePeriodInSeconds,
            _icoStageID: _icoStageID,
            _stakingEndTime: startTime - 1
        });
    }
```

Recommendation

It is recommended to introduce an upper bound check for the `timePeriodInSeconds` parameter to ensure that the staking period fits within the intended window before the associated vesting schedule begins. Specifically, the time period should be validated to fall within the range between the current timestamp and the defined staking end time to maintain consistency with the contract's temporal logic and avoid unintended staking configurations.

UPU - Unnecessary Parameter Usage

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L345,396 ex1EthICO.sol#L202
Status	Unresolved

Description

The contract is using a parameter `_token` in functions where the contract already has predefined token addresses. Since the contract knows which token is used in each function, there is no need to pass the `_token` parameter, and doing so introduces unnecessary complexity and potential confusion.

```
    function buyWithUSDC(
         uint256 _amount,
         uint256 _icoStageID,
         IERC20 _token
     ) external checkSaleStatus(_icoStageID) returns(bool) {
         require(
             _token == USDCAddress,
             "ex1Sale: Token Invalid!"
         );
         ...
    function buyWithUSDT(
         uint256 _amount,
         uint256 _icoStageID,
         IERC20 _token
     ) external checkSaleStatus(_icoStageID) returns(bool) {
         require(
             _token == USDTAddress,
             "ex1Sale: Token Invalid!"
         );
         ...
         }
```

```
   function purchasedViaEth(
        uint256 _amount,
        uint256 _icoStageID
    ) external checkSaleStatus(_icoStageID) payable nonReentrant {
        require(
            block.timestamp >= icoStages[_icoStageID].startTime &&
block.timestamp <= icoStages[_icoStageID].endTime,
            "ex1Presale: Invalid Stage Paramaters"
        );
        uint256 ethAmount = getTokenPriceInETH(_amount, _icoStageID);
        require(
            msg.value >= ethAmount,
            "EthPayment: Insufficient Eths Value signed!"
        );
        ...
```

Recommendation

It is recommended to remove the `_token` parameter from these functions and rely on the contract's existing knowledge of the predefined token addresses. This simplification will reduce complexity, enhance clarity, and eliminate the need for redundant checks against a parameter that the contract can already determine.

UEE - Unrefunded Excess ETH

Criticality	Minor / Informative
Location	ex1EthICO.sol#L202
Status	Unresolved

Description

The `purchasedViaEth` function does not handle refunding excess `msg.value` to the sender. If the `msg.value` provided exceeds the required ETH amount, the surplus remains unreturned, effectively resulting in users overpaying without compensation. This behavior may lead to a poor user experience, as users cannot reclaim their overpaid funds.

```
 function purchasedViaEth(
        uint256 _amount,
        uint256 _icoStageID
    ) external checkSaleStatus(_icoStageID) payable nonReentrant {
        ...

        (bool success, ) = payable(receivingWallet).call{value:
msg.value}("");
        require(
            success,
            "Ex1 ETH: Transfer of ETH failed"
        );

        ...
    }
```

Recommendation

It is recommended to implement logic that calculates and refunds the difference between `msg.value` and the required ETH amount. By ensuring that any excess funds are promptly returned to the user, the contract will maintain a fair and user-friendly transaction process.

UVS - Unused Variable Set

Criticality	Minor / Informative
Location	ex1EthICO.sol#L66,254
Status	Unresolved

Description

The `MaxTokenLimitPerAddress` variable is set via the `setMaxTokenLimitPerAddress` function but is not utilized anywhere else within the contract. Although it is stored as a public state variable, it does not appear in any other logic, calculations, or conditions throughout the contract. This unused variable contributes no functional value and may cause confusion or lead to unnecessary storage costs.

```
    uint256 public MaxTokenLimitPerAddress;

     function setMaxTokenLimitPerAddress(
         uint256 _limit
     ) external onlyRole(OWNER_ROLE) {
         MaxTokenLimitPerAddress = _limit;
     }
```

Recommendation

It is recommended to either remove the `MaxTokenLimitPerAddress` variable and its setter function if they are not needed, or to integrate it into the contract logic if it is intended to serve a specific purpose. This change will eliminate unnecessary code, reduce storage usage, and maintain a cleaner, more efficient codebase.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	ex1Token.sol#L106,107,138,149,165,173,182,212,246,273,315,352 ex1Staking.sol#L78,79,80,112,142,171,172,246,247,303,304,315,331,335,339 ex1PrivateVesting.sol#L145,146,147,148,149,150,151,152,226,227,228,229,230,231,232,233,277,313,347,373,397,411,419 ex1ICOVesting.sol#L30,90,91,92,93,94,136,137,138,139,140,174,224,225,265,266,300,301,316,324 ex1ICOv2.sol#L19,20,38,39,55,56,57,59,60,135,136,137,138,236,237,238,239,293,308,323,324,346,347,348,397,398,399,455,456,457,458,508,509,510,511,554,560,566,571,583,591,599,607 ex1EthICO.sol#L66,67,75,76,77,112,113,114,115,116,145,146,147,148,171,186,187,203,204,240,249,255,260
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[] memory _approver
uint256 _required
address _address
uint256 _value
address _to
uint256 _txIndex
uint256 _percentageReturn
uint256 _timePeriodInSeconds
uint256 _icoStageID
address _caller
Iex1ICO _icoInterface
IVestingICO _vestingInterface
IERC20 _tokenAddress
address _beneficiary

...
```

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.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L176,259 ex1ICOVesting.sol#L110,158
Status	Unresolved

Description

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

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

```
require(
            _slicePeriod >= 0 && _slicePeriod <= 60,
            "PrivateVesting: Invalid Slice Period"
        )

require(
            _slicePeriod >= 0 && _slicePeriod <= 60,
            "ex1Presale: Invalid Slice Period"
        )
```

Recommendation

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

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	ex1Token.sol#L213,253,254,255,256,279,280,281,282 ex1Staking.sol#L179,196,200,254,271,274 ex1ICOv2.sol#L206,213
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
isAddressRestricted(_msgSender()) == true
require(isApprover[_msgSender()] == true, "Approver Status Held!")
require(isRevoked[_txIndex] == false, "Transaction status Revoked!")
require(transactions[_txIndex].executed == false, "Transaction already
Executed")
require(hasVoted[_txIndex][_msgSender()] == false, "Approver already
Voted")

uint256 deposits = unstaked[_icoStageID][_caller] == true
        ? formerTotalStakedPerICO[_icoStageID][_caller]
        : totalStakedPerICO[_icoStageID][_caller]
unstaked[_icoStageID][_caller] == true && block.timestamp >
stakingEndTime
unstaked[_icoStageID][_caller] == true

(block.timestamp > icoStages[_id].startTime) &&
                (block.timestamp < icoStages[_id].endTime) &&
                (icoStages[_id].isActive == true)

...
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	ex1Staking.sol#L188,212,217,225,232,263,283,287,292,296 ex1PrivateVesting.sol#L330,337,339 ex1ICOVesting.sol#L244,251,253
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 userRewardPerSecond = userPercentage /
stakingParameters[_icoStageID].timePeriodInSeconds
reward = ((time - stakeTimestamp[_icoStageID][_caller]) *
userRewardPerSecond)

uint256 tokenPerSlice = schedule.totalAmount / totalNumberOfSlices
elapsedSlices = (block.timestamp -
lastClaimedTimestamp[_vestingScheduleID])/ schedule.slicePeriod
uint256 claimable = tokenPerSlice * elapsedSlices

uint256 tokenPerSlice = totalDeposits / totalNumberOfSlices
elapsedSlices = (block.timestamp -
prevClaimTimestamp[_icoStageID][_caller])/ schedule.slicePeriod
uint256 claimable = tokenPerSlice * elapsedSlices -
claimedAmount[_icoStageID][_msgSender()]
```

Recommendation

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

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
EX1	Implementation	Initializable, ERC20Upgradeable, AccessControlUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	updateApprovers	External	✓	onlyRole
	addApprover	External	✓	onlyRole
	addRestrictedAddress	External	✓	onlyRole
	removeRestrictedAddress	External	✓	onlyRole
	isAddressRestricted	Public		-
	checkRestrictedAddress	Public		-
	checkApprovers	Public		-
	transfer	Public	✓	-
	_proposeTransfer	Internal	✓	
	approveTransfer	Public	✓	onlyRole txExists notExecuted notVoted
	revokeApproval	Public	✓	onlyRole txExists notExecuted
	_executeTransfer	Internal	✓	txExists notExecuted

	executeTransfer	External	✓	txExists notExecuted onlyRole onlyRole
	getAllPendingProposals	External		-
	updateRequiredApprovers	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole
Ex1Staking	Implementation	Initializable, ReentrancyGuardUpgradeable, AccessControlUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	createStakingRewardsParamaters	External	✓	onlyRole
	stake	External	✓	-
	claimStakingRewards	External	✓	nonReentrant
	calculateStakeReward	Internal	✓	
	viewClaimableRewards	External		-
	getEligibleStakableToken	Public		-
	unstake	External	✓	-
	updateIcoInterface	External	✓	onlyRole
	updateVestingInterface	External	✓	onlyRole
	updateEX1Token	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole

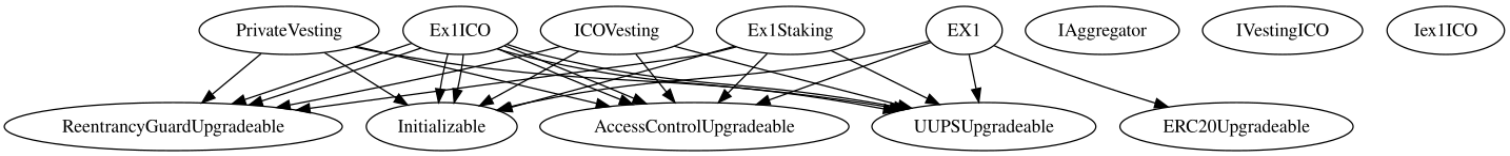
PrivateVesting	Implementation	Initializable, AccessControlUpgradeable, ReentrancyGuardUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	setVestingSchedule	External	✓	onlyRole
	updateVesting	External	✓	onlyValidSchedule onlyRole
	claimTokens	External	✓	nonReentrant onlyValidSchedule notRevoked
	calculateClaimableAmount	Public		onlyValidSchedule notRevoked
	revokeSchedule	External	✓	onlyRole onlyValidSchedule notRevoked
	nextClaimTime	Public		onlyValidSchedule notRevoked
	getBalanceLeftToClaim	Public		-
	getAllVestingSchedules	Public		-
	getBeneficiarySchedules	External		-
	setEx1TokenSaleContract	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole
Ex1ICO	Implementation	Initializable, ReentrancyGuardUpgradeable, AccessControlUpgradeable, UUPSUpgradeable		

		Public	✓	-
	initialize	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyRole
	createICOStage	External	✓	onlyRole
	getAllICOStages	External		-
	nextICOStageID	External		-
	getCurrentOrNextActiveICOStage	External		-
	updateICOStage	External	✓	onlyRole
	getLatestETHPrice	Public		-
	getLatestBTCPrice	Public		-
	getTokenPriceInETH	Public		-
	getTokenPriceInBTC	External		-
	calculatePrice	Public		-
	buyWithUSDC	External	✓	checkSaleStatus
	buyWithUSDT	External	✓	checkSaleStatus
	purchasedViaEth	External	✓	checkSaleStatus onlyRole
	purchasedViaBTC	External	✓	checkSaleStatus onlyRole
	setMaxTokenLimitPerAddress	External	✓	onlyRole
	setTokenSaleAddress	External	✓	onlyRole
	setMaxTokenLimitPerTransaction	External	✓	onlyRole
	setReceiverWallet	External	✓	onlyRole
	setTokenReleasable	External	✓	onlyRole

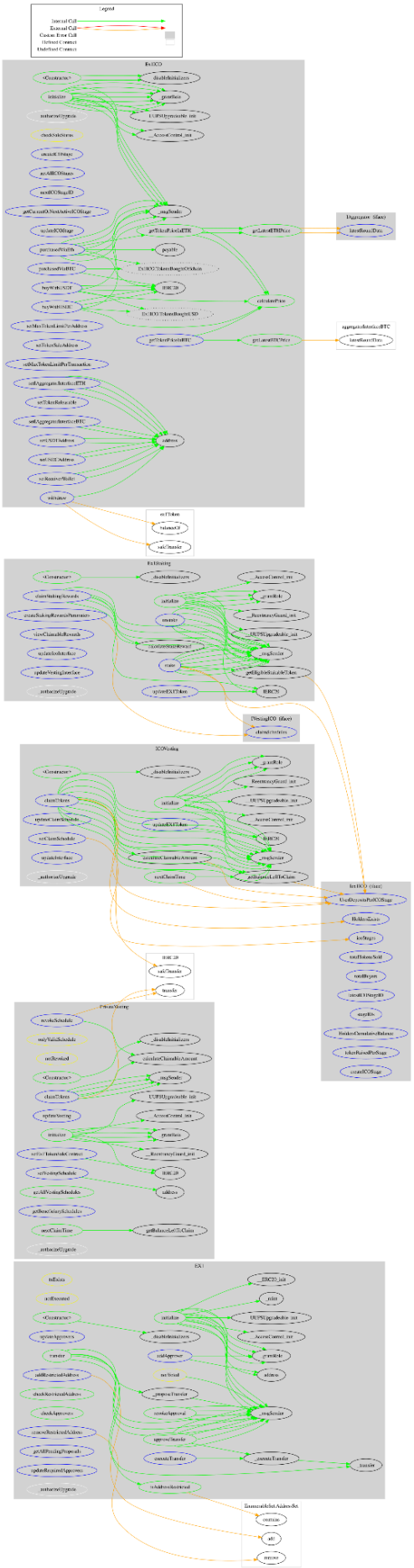
	setIAggregatorInterfaceETH	External	✓	onlyRole
	setIAggregatorInterfaceBTC	External	✓	onlyRole
	setUSDTAddress	External	✓	onlyRole
	setUSDCAddress	External	✓	onlyRole
	withdraw	External	✓	onlyRole
ICOVesting	Implementation	Initializable, AccessControlUpgradeable, ReentrancyGuardUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	setClaimSchedule	External	✓	onlyRole
	updateClaimSchedule	External	✓	onlyRole
	claimTokens	External	✓	nonReentrant
	calculateClaimableAmount	Public		-
	nextClaimTime	Public		-
	getBalanceLeftToClaim	Public		-
	updateInterface	External	✓	onlyRole
	updateEX1Token	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole
IAggregator	Interface			
	latestRoundData	External		-

Ex1ICO	Implementation	Initializable, ReentrancyGuardUpgradeable, AccessControlUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	createICOStage	External	✓	onlyRole
	updateICOStage	External	✓	onlyRole
	getLatestETHPrice	Public		-
	getTokenPriceInETH	Public		-
	calculatePrice	Public		-
	purchasedViaEth	External	Payable	checkSaleStatus nonReentrant
	setReceiverWallet	External	✓	onlyRole
	setMaxTokenLimitPerTransaction	External	✓	onlyRole
	setMaxTokenLimitPerAddress	External	✓	onlyRole
	setIAggregatorInterfaceETH	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole

Inheritance Graph



Flow Graph



Summary

The eXchange1 suite of smart contracts implements a comprehensive, role-based, and upgradeable ecosystem for secure EX1 token issuance, cross-chain ICO sales, linear vesting, staking, and private allocations, leveraging OpenZeppelin libraries and price feeds. This audit investigates security vulnerabilities, business logic inconsistencies, and potential optimizations across the EX1, Ex1ICO, Ex1EthICO, ICOVesting, Ex1Staking, and PrivateVesting contracts to ensure robust and transparent token management.

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