Cyberscope

Audit Report

eXchange1

June 2025

Repository	https://github.com/ex1ico/ex1SmartContracts
Commit	284bf924d31a71c4c30c1a5d286715e48b09aa16
Audited by	© cyberscope

Table of Contents

Risk Classification

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

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

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

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

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

Review

Repository	https://github.com/ex1ico/ex1SmartContracts
Commit	284bf924d31a71c4c30c1a5d286715e48b09aa16

Audit Updates

Initial Audit	18 Apr 2025
Corrected Phase 2	13 Jun 2025

Source Files

Filename	SHA256
ex1Staking.sol	c1574688ab9bfb9c08311a5c4714d06d6012515ea53f52b402f6e5b5fa13c31e
ex1PrivateVesting.sol	b24f8e40382aa5e5b01937d7a0dc372afbb98e9cd2c47c93d18192d239d02c46
ex1ICOv2.sol	9942bd65c1f17baf6ee4f996efa31e567e031bfd710fbc80969c31717065ba30
ex1ICOVesting.sol	2a956fba311a26bc78cd452fe282037a6ad45be2021a44f8731b14bc01ecf769
ex1EthICO.sol	43772c38369c88c2c04027ca29c6d6d55f9f0247a452dfabd3deb71b88b849e6
eX1token.sol	0c6d02e31744526db3fb2aa653dfaadcb800b4b5f09d6775013181448ab4d6fa

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.

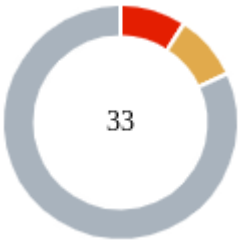
ex1EthICO file

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	3
Minor / Informative	27

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	3	0	0	0
Medium	3	0	0	0
Minor / Informative	9	18	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IEF	Incorrect ETH Forwarding	Unresolved
●	MAV	Missing Allowance Validation	Unresolved
●	PUF	Premature Unstake Flag	Unresolved
●	ACI	Approval Count Imbalance	Unresolved
●	IETC	Incorrect Elapsed Time Calculation	Unresolved
●	MRR	Missing Role Revocation	Unresolved
●	BC	Blacklists Addresses	Acknowledged
●	CR	Code Repetition	Acknowledged
●	CCR	Contract Centralization Risk	Acknowledged
●	IDU	Inconsistent Data Updates	Acknowledged
●	ISR	Inconsistent Stake Reset	Unresolved
●	IEC	Incorrect ETH Calculations	Acknowledged
●	ITN	Incorrect Token Name	Unresolved
●	IVS	Inefficient Vesting Setup	Acknowledged

●	MCIC	Missing Claim Interval Check	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MEE	Missing Events Emission	Acknowledged
●	MEC	Missing Existence Check	Acknowledged
●	MSEC	Missing Stage Existence Check	Unresolved
●	MTDC	Missing Token Decimal Check	Acknowledged
●	MUV	Missing User Validation	Acknowledged
●	MU	Modifiers Usage	Acknowledged
●	ODM	Oracle Decimal Mismatch	Acknowledged
●	OSET	Overlapping Start End Times	Acknowledged
●	POSD	Potential Oracle Stale Data	Acknowledged
●	PTRP	Potential Transfer Revert Propagation	Acknowledged
●	RDC	Redundant Duration Check	Unresolved
●	RTL	Redundant Transfer Logic	Unresolved
●	SBI	Staking Before Initialization	Unresolved
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	UIC	Unreachable If Condition	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Acknowledged

●	L13	Divide before Multiply Operation	Acknowledged

IEF - Incorrect ETH Forwarding

Criticality	Critical
Location	ex1EthICO.sol#L252
Status	Unresolved

Description

The contract is forwarding the entire `msg.value` to the receiving wallet before attempting to refund any excess ETH to the sender. However, since the full amount has already been transferred out, the contract no longer retains any ETH to process the refund. As a result, when the contract tries to refund the `excessEth` to the sender, it lacks the required balance, causing the transaction to revert. This makes all such buy transactions fail, even if they would have otherwise been valid.

```
(bool success, ) = payable(receivingWallet).call{value: msg.value}("");
require(success, "Ex1 ETH: Transfer of ETH failed")
uint256 excessEth = msg.value - ethAmount;
if (excessEth > 0) {
    (bool refunded, ) = payable(_msgSender()).call{value: excessEth}(
        ""
    );
    require(refunded, "Ex1 ETH: Transfer of excess ETH failed");
}
```

Recommendation

It is recommended to send only the required `ethAmount` to the receiving wallet and retain the excess ETH within the contract until the refund is processed. This ensures the contract maintains sufficient balance to execute the refund and avoids unnecessary transaction failures.

MAV - Missing Allowance Validation

Criticality	Critical
Location	ex1Token.sol#232
Status	Unresolved

Description

The contract is implementing a `transferFrom` function that bypasses the standard ERC-20 allowance check mechanism. Specifically, it does not verify whether the caller has been granted sufficient allowance to transfer tokens on behalf of `_from`. This omission deviates from the expected behaviour defined in the ERC-20 standard and will allow unauthorised token transfers, especially in contexts where dApps or users rely on allowance-based permissioning for delegated spending.

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

Recommendation

It is recommended to implement an explicit allowance check within the `transferFrom` function to ensure the caller is permitted to spend the specified amount on behalf of the token owner. Additionally, the allowance should be decreased accordingly after a successful transfer to align with ERC-20 standards and prevent potential misuse.

PUF - Premature Unstake Flag

Criticality	Critical
Location	ex1Staking.sol#L135,244
Status	Unresolved

Description

The contract is incorrectly flagging the user as unstaked before calculating their staking reward. Specifically, the `unstake` function sets `unstaked[_icoStageID][_msgSender()] = true` prior to invoking `calculateStakeReward`. As a result, the `calculateStakeReward` function interprets the user as already unstaked and sets `deposits` to zero, which leads to a revert due to the `require(deposits > 0)` check. This flaw prevents users from successfully unstaking and claiming their rewards, effectively blocking the intended functionality of the contract.

```
    function calculateStakeReward(uint256 _icoStageID, address _caller)
internal returns (uint256) {
        uint256 deposits = unstaked[_icoStageID][_caller] ? 0 :
totalStakedPerICO[_icoStageID][_caller];
        require(deposits > 0, "ex1Staking: No Tokens Staked");
        ...

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

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

        uint256 reward = calculateStakeReward(_icoStageID, _msgSender());
        require(reward > 0, "ex1Staking: No Rewards to Claim!");
        ...
        return true;
    }
```

Recommendation

It is recommended to defer setting the `unstaked` flag until after reward calculation has been completed successfully. This ensures that the `calculateStakeReward` function computes rewards based on the user's actual staked amount prior to marking them as unstaked.

ACI - Approval Count Imbalance

Criticality	Medium
Location	ex1Token.sol#L291
Status	Unresolved

Description

The contract is not correctly maintaining the `approvalCount` for transactions when an approval is revoked. While the `approvalCount` is incremented each time an approver grants approval, it is never decremented when an approver revokes their approval using the `revokeApproval` function. This leads to a state inconsistency where the recorded number of approvals no longer reflects the actual number of valid approvals, potentially causing logical errors in approval-based execution checks or misrepresenting consensus among approvers.

```
   function revokeApproval(uint256 _txIndex) public
onlyRole(APPROVER_ROLE) txExists(_txIndex) notExecuted(_txIndex) {
        require(!isRevoked[_txIndex], "Transaction status Revoked!");
        require(isApprovedBy[_txIndex][_msgSender()], "Signer needs to
Approve first!");
        require(!isRevokedBy[_txIndex][_msgSender()], "Signer already
Revoked!");

        transactions[_txIndex].revokeCount += 1;
        isRevokedBy[_txIndex][_msgSender()] = true;

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

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

Recommendation

It is recommended to decrement the `approvalCount` within the `revokeApproval` function when an approver successfully revokes their approval. This ensures the contract maintains accurate tracking of active approvals, preserving the correctness and reliability of the multi-signature or approval threshold logic.

IETC - Incorrect Elapsed Time Calculation

Criticality	Medium
Location	ex1PrivateVesting.sol#L312
Status	Unresolved

Description

The contract calculates the elapsed time for vesting based on the `cliffTime` instead of the `startTime`. While the cliff is meant to enforce a delay before any tokens become claimable, it should not be used as the reference point for calculating the vesting progress. Vesting is typically designed to begin at the `startTime`, with the `cliffTime` acting as a gate to prevent any claims before a specified duration has passed. By incorrectly using `cliffTime` as the basis for the elapsed calculation, the contract skews the reward distribution and delays the vesting schedule unintentionally, resulting in incorrect claimable amounts.

```
uint256 elapsed = (block.timestamp - cliffTime) / schedule.slicePeriod;
if (elapsed > totalSlices) {
    elapsed = totalSlices;
}
```

Recommendation

It is recommended to calculate the elapsed time using `startTime` as the reference point for vesting progression. The `cliffTime` should only serve as a condition to block any claims until it is reached. This ensures that once the cliff period is passed, the claimable amount reflects the correct proportion of time that has elapsed since the actual start of vesting.

MRR - Missing Role Revocation

Criticality	Medium
Location	eX1token.sol#L126
Status	Unresolved

Description

The contract is improperly managing role-based permissions when updating approvers. Specifically, the `updateApprovers` function modifies the internal `isApprover` mapping based on the new status but does not revoke the `APPROVER_ROLE` from addresses that are being deactivated. As a result, an address may retain elevated privileges associated with the `APPROVER_ROLE` even after being marked as inactive, which poses a risk of unauthorized access to functions gated by that role and undermines the integrity of the role-based access control mechanism.

```
    function updateApprovers(address[] memory _approver, bool[] memory
status) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_approver.length == status.length, "Length Mismatched!");
        for (uint256 i = 0; i < _approver.length; i++) {
            bool isApproverStatus = checkApprovers(_approver[i]);
            require(isApproverStatus, "Approver not found!");
            isApprover[_approver[i]] = status[i];
        }
    }

    function addApprover(address[] memory _approver) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        for (uint256 i = 0; i < _approver.length; i++) {
            address approver = _approver[i];
            require(approver != address(0), "invalid approver");
            require(!isApprover[approver], "approver not unique");

            isApprover[approver] = true;
            _grantRole(APPROVER_ROLE, approver);
            approvers.push(approver);
        }
    }
```

Recommendation

It is recommended to enhance the `updateApprovers` function by including logic to revoke the `APPROVER_ROLE` from addresses whose status is being set to inactive. This will ensure that role permissions remain tightly coupled to the current access status and reduce the risk of unintended privilege retention.

BC - Blacklists Addresses

Criticality	Minor / Informative
Location	ex1Token.sol#L155,216
Status	Acknowledged

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;
    }

    function transferFrom(address _from, address _to, uint256 _value)
public virtual override returns (bool) {
        if (isAddressRestricted(_from) == true) {
            _proposeTransfer(_from, _to, _value);
        } else {
            _transfer(_from, _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#L362,404
Status	Acknowledged

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 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	Acknowledged

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.

IDU - Inconsistent Data Updates

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L199 ex1EthICO.sol#L122 ex1ICOVesting.sol#L173 ex1Staking.sol#L100 ex1PrivateVesting.sol#L258
Status	Acknowledged

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.

ISR - Inconsistent Stake Reset

Criticality	Minor / Informative
Location	ex1Staking.sol#L174,254
Status	Unresolved

Description

The contract is inconsistently resetting staking records across different staking completion scenarios. Specifically, while the `totalStakedPerICO` mapping is correctly reset during the `unstake` process, the `calculateStakeReward` function fails to reset this value in its final `else` case—where all staking time has already passed. As a result, the user's `totalStakedPerICO` remains non-zero despite being flagged as unstaked. This leads to data inconsistency across the contract's internal tracking structures, potentially affecting reward calculations, eligibility checks, or any logic that relies on the actual staked amount.

```
 function calculateStakeReward(uint256 _icoStageID, address _caller)
internal returns (uint256) {
        ...
            } else {
                reward =
                    ((stakingEndTime -
previousStakingRewardClaimTimestamp[_icoStageID][_caller]) *
userRewardPerSecond);
                previousStakingRewardClaimTimestamp[_icoStageID][_caller]
= block.timestamp;
                isStaked[_icoStageID][_caller] = false;
                unstaked[_icoStageID][_caller] = true;
                return reward;
            }
        }
    }

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

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

        uint256 reward = calculateStakeReward(_icoStageID, _msgSender());
        require(reward > 0, "ex1Staking: No Rewards to Claim!");

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

Recommendation

It is recommended to ensure that `totalStakedPerICO` is consistently reset in all staking completion paths, including within the final branch of the `calculateStakeReward` function. This will maintain internal state consistency and ensure that staking data accurately reflects the user's active or completed staking status across all conditions.

IEC - Incorrect ETH Calculations

Criticality	Minor / Informative
Location	ex1EthICO.sol#L149
Status	Acknowledged

Description

The contract contains a logic in the `getTokenPriceInETH` function, which determines the ETH equivalent for purchasing tokens. The issue stems from improper handling of decimal precision and inconsistent scaling between the ETH price and the USD-denominated token price. If the ETH price is not aligned with the expected scale used in the token pricing logic, it results in inaccurate conversions. This mismatch can significantly distort the ETH amount required for purchases, potentially leading to users being overcharged and undermining the reliability of the token sale mechanism.

This means that the ETH price must be in a specific scaling format that matches the USD value calculation to ensure the token-to-ETH conversion is accurate.

```
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.

Team Update

The team has acknowledged that this is not a security issue and states:

The _tokenPriceUSD will follow 10^18 format (with 18 decimals). The issue was raised due to 10^6 usage; using 10^18 resolves the concern as intended.

ITN - Incorrect Token Name

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L285,301,320
Status	Unresolved

Description

The contract is designed to allow token purchases using either USDC or USDT by passing the respective token address and a hardcoded name string to an internal function. However, this setup permits a user to call `buyWithUSDC` while providing the address of the USDT token, or vice versa. In such cases, the event emitted (`TokensBoughtUSD`) will include an incorrect token name, as the name is based on the function called rather than the actual token used. This leads to misleading event logs, which may affect off-chain monitoring, auditing, or user interfaces that depend on accurate event data.

```
    function buyWithUSDC(uint256 _amount, uint256 _icoStageID, IERC20
_token)
        external
        checkSaleStatus(_icoStageID)
        returns (bool)
    {
        string memory _usdName = "USDC";
        bool success = _buyWithUSD(_amount, _icoStageID, _token,
_usdName);
        return success;
    }

    function buyWithUSDT(uint256 _amount, uint256 _icoStageID, IERC20
_token)
        external
        checkSaleStatus(_icoStageID)
        returns (bool)
    {
        string memory _usdName = "USDT";
        bool success = _buyWithUSD(_amount, _icoStageID, _token,
_usdName);
        return success;
    }

 function _buyWithUSD(uint256 _amount, uint256 _icoStageID, IERC20
_token, string memory _usdName)
        internal
        checkSaleStatus(_icoStageID)
        returns (bool)
    {
        require(_token == USDCAddress || _token == USDTAddress, "ex1Sale:
Token Invalid!");
        uint256 usdValue = calculatePrice(_amount, _icoStageID);

        ...
        return true;
    }
```

Recommendation

It is recommended to enforce a strict correlation between the token address and its associated name within the contract logic. Specifically, the contract should internally determine the correct token name based on the provided token address rather than relying on user-defined or function-specific naming, thereby ensuring accurate and trustworthy event emissions.

IVS - Inefficient Vesting Setup

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L147
Status	Acknowledged

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.

Team Update

The team has acknowledged that this is not a security issue and states:

Allocation targets specific wallets (treasury, liquidity, marketing), only ~8 beneficiaries.

MCIC - Missing Claim Interval Check

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L160
Status	Unresolved

Description

The contract is missing a validation check in the `setVestingSchedule` function to prevent the `_claimInterval` from being set to zero. While the `updateVesting` function includes a condition that ensures `_claimInterval` is greater than zero, this safeguard is not present in `setVestingSchedule`. 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) {
        ...ivateVesting: Invalid Cliff Period"
        );
        require(
            _claimInterval <= (_endTime - _cliffPeriod),
            "Private: Invalid Claim Interval"
        );
        ...

        vestingSchedules[latestVestingScheduleID] = VestingSchedule({
            beneficiary: _beneficiary,
            vestingScheduleID: latestVestingScheduleID,
            totalAmount: _totalAmount,
            startTime: _startTime,
            endTime: _endTime,
            claimInterval: _claimInterval,
            ...
```

Recommendation

It is recommended to add a validation step in the `setVestingSchedule` 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.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	ex1EthICO.sol#L112
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.

Specifically, the `createICOStage` function emits the update `ICOStageUpdate` event instead of a creation event.

```
    function createICOStage(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        uint256 _icoStageID,
        bool _isActive
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        ...

        emit ICOStageUpdate(_icoStageID, _startTime, _endTime,
_tokenPriceUSD, _isActive);
    }
```

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	ex1Token.sol#L126 ex1ICOv2.sol#L451 ex1PrivateVesting.sol#L207
Status	Acknowledged

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	ex1ICOVesting.sol#L116
Status	Acknowledged

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) {

        ...
    }
```

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.

MSEC - Missing Stage Existence Check

Criticality	Minor / Informative
Location	ex1EthICO.sol#L162
Status	Unresolved

Description

The contract is implementing an `updateICOStage` function that does not validate whether the specified `_stageID` corresponds to an existing ICO stage before applying updates. Unlike the `createICOStage` function, which includes a check to prevent overwriting an existing stage, the update function assumes the stage already exists. If the provided `_stageID` does not correspond to a previously created stage, the update will silently write to an uninitialized storage slot, potentially introducing invalid or unintended data into the system.

```
  function createICOStage(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _tokenPriceUSD,
        uint256 _icoStageID,
        bool _isActive
    ) external onlyRole(ICO_AUTHORISER_ROLE) {
        ...

        stageIdExists[_icoStageID] = true;
        stageIDs.push(_icoStageID);

       ...
    }


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

Recommendation

It is recommended to include a `require` check in the `updateICOStage` function to verify that the given `_stageID` has been previously created (e.g., by checking `stageIdExists[_stageID] == true`). This ensures updates are only made to valid, existing stages and prevents accidental creation of undefined or inconsistent stage entries.

MTDC - Missing Token Decimal Check

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L589
Status	Acknowledged

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#L347
Status	Acknowledged

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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	ex1Token.sol#L272 ex1ICOv2.sol#L130,461 ex1PrivateVesting.sol#L93 ex1Staking.sol#L120
Status	Acknowledged

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(!isRevoked[_txIndex], "Transaction status Revoked!");
require(!hasVoted[_txIndex][_msgSender()], "Approver already Voted");
require(!isApprovedBy[_txIndex][_msgSender()], "Approver already
Approved");
```

```
require((_startTime < _endTime), "ex1Presale: Invalid Schedule or
Parameters!");
require(_startTime > block.timestamp, "ex1Presale: Invalid Start Time!");
require(_endTime > block.timestamp, "ex1Presale: Invalid End Time!");

...
require(
    address(_aggregator) != address(0),
    "ex1Presale: Invalid Aggregator Address!"
);
```

```
require(_startTime > endTime, "ex1Presale: Token Sale not Ended yet!");

...
```

```
`require(isStaked[_icoStageID][_msgSender()], "ex1Staking: Not Staked Yet!");
```

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#L139
Status	Acknowledged

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#L199
Status	Acknowledged

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#L139
Status	Acknowledged

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#L192
Status	Acknowledged

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.

RDC - Redundant Duration Check

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L306
Status	Unresolved

Description

The contract includes a conditional assignment: `uint256 vestingDuration = vestingEnd > cliffTime ? vestingEnd - cliffTime : 0;`. However, the variables `vestingEnd` and `cliffTime` are set such that `vestingEnd` is always greater than `cliffTime` by design. As a result, the condition `vestingEnd > cliffTime` will always evaluate to `true`, rendering the ternary check redundant. This introduces unnecessary complexity to the code without adding any functional value or safety check, making the intent less clear to readers or future maintainers.

```
uint256 vestingDuration = vestingEnd > cliffTime ? vestingEnd -
cliffTime : 0;
```

Recommendation

It is recommended to simplify the expression by directly assigning `vestingDuration = vestingEnd - cliffTime;`, assuming the contract's logic guarantees `vestingEnd` is always greater than `cliffTime`. If this assumption is ever violated, consider adding an explicit `require` check to validate this condition.

RTL - Redundant Transfer Logic

Criticality	Minor / Informative
Location	ex1Token.sol#L216
Status	Unresolved

Description

The contract is implementing both `transfer` and `transferFrom` functions with duplicated logic, where each function checks if an address is restricted and then either calls `_proposeTransfer` or `_transfer`. This repeated pattern introduces unnecessary code duplication, which can increase maintenance overhead and the risk of inconsistencies if one function is updated without reflecting the change in the other.

```
    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;
    }


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

Recommendation

It is recommended to abstract the shared conditional logic into a single internal function that accepts the sender, receiver, and amount as parameters. Both `transfer` and `transferFrom` should delegate to this internal function, thereby improving maintainability, reducing duplication, and ensuring consistent execution behaviour.

SBI - Staking Before Initialization

Criticality	Minor / Informative
Location	ex1Staking.sol#L75,100
Status	Unresolved

Description

The contract is allowing users to stake tokens for a specific ICO stage before the staking parameters are explicitly initialized via the `createStakingRewardsParamaters` function. This creates a critical race condition where staking actions can be performed without defined parameters such as the return percentage, time period, or staking end time. As a result, users may lock their funds into an undefined or partially configured staking process, leading to inconsistent reward calculations, unintended staking durations, or potential loss of expected returns.

```
    function createStakingRewardsParamaters(
        uint256 _percentageReturn,
        uint256 _timePeriodInSeconds,
        uint256 _icoStageID
    ) external onlyRole(STAKING_AUTHORISER_ROLE) {
        (, uint256 startTime,,,) =
vestingInterface.claimSchedules(_icoStageID);
        require(_percentageReturn > 0 && _percentageReturn <= 100,
"ex1Staking: Invalid Percentage!");
        require(
            _timePeriodInSeconds > 0 && block.timestamp +
_timePeriodInSeconds < startTime,
            "ex1Staking: Invalid Time Period!"
        );
        require(startTime > 0, "ex1Staking: Vesting Schedule not set for
this ICO stage!");
        require(startTime > block.timestamp, "ex1Staking: Vesting
Schedule Claiming already Initiated");
        stakingParameters[_icoStageID] = StakingParamter({
            percentageReturn: _percentageReturn,
            timePeriodInSeconds: _timePeriodInSeconds,
            _icoStageID: _icoStageID,
            _stakingEndTime: startTime - 1
        });
    }


    function stake(uint256 _icoStageID) external returns (bool) {
        uint256 deposits =
icoInterface.UserDepositsPerICOStage(_icoStageID, _msgSender());
        (, uint256 startTime,,,) =
vestingInterface.claimSchedules(_icoStageID);
        require(getEligibleStakableToken(_icoStageID, _msgSender()) > 0,
"ex1staking: No Balance available to stake");
        require(deposits > 0, "ex1Staking: No Tokens to Stake!");
        require(block.timestamp <= startTime, "ex1Staking: Staking Not
Active!");

        isStaked[_icoStageID][_msgSender()] = true;
        totalStakedPerICO[_icoStageID][_msgSender()] = deposits;
        stakeTimestamp[_icoStageID][_msgSender()] = block.timestamp;
        previousStakingRewardClaimTimestamp[_icoStageID][_msgSender()] =
0;
        unstaked[_icoStageID][_msgSender()] = false;
        return true;
    }
```

Recommendation

It is recommended to enforce a check within the `stake` function to ensure that staking parameters have been properly initialized for the given ICO stage before allowing staking actions. This will guarantee that all critical conditions and reward logic are fully defined and in place, preventing premature or misconfigured staking interactions.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L372,414 ex1ICOVesting.sol#L170 ex1Staking.sol#L123,256 ex1Token.sol#L330
Status	Acknowledged

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.

UIC - Unreachable If Condition

Criticality	Minor / Informative
Location	ex1ICOVesting.sol#L179 ex1PrivateVesting.sol#L291
Status	Unresolved

Description

The contract includes an `if` condition that checks whether the number of elapsed vesting slices is equal to the total number of slices (`if (elapsedSlices == totalNumberOfSlices)`) in order to unlock the full deposit. However, this condition is logically unreachable due to a preceding check: `if (block.timestamp >= schedule.endTime)` , which is triggered before the slice comparison is evaluated. Since reaching the total number of slices implies that the current timestamp is equal to or beyond the schedule's `endTime` , the earlier `if` condition will always be satisfied first. This makes the later check redundant and introduces ambiguity about the intended logic, potentially leading to confusion or maintenance issues in future updates.

```
function calculateClaimableAmount(address _caller, uint256 _icoStageID)
public view returns (uint256) {
        ...

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

        if (block.timestamp >= schedule.endTime) {
            return totalDeposits - claimedAmount[_icoStageID][_caller];
        }

        uint256 elapsedSlices = prevClaimTimestamp[_icoStageID][_caller]
== 0
            ? (block.timestamp - schedule.startTime) /
schedule.slicePeriod
            : (block.timestamp -
prevClaimTimestamp[_icoStageID][_caller]) / schedule.slicePeriod;

        uint256 unlocked = totalDeposits * elapsedSlices /
totalNumberOfSlices;

        if (elapsedSlices == totalNumberOfSlices) {
            unlocked = totalDeposits;
        }

        return unlocked - claimedAmount[_icoStageID][_caller];
    }
```

```
   function calculateClaimableAmount(uint256 _vestingScheduleID)
        public
        view
        onlyValidSchedule(_vestingScheduleID)
        notRevoked(_vestingScheduleID)
        returns (uint256)
    {
        ...
        if (elapsed == totalSlices) {
            unlocked = schedule.totalAmount;
        }

        return unlocked - schedule.releasedAmount;
    }
```

Recommendation

It is recommended to reconsider the code structure and remove the unreachable `if (elapsedSlices == totalNumberOfSlices)` condition. This will simplify the logic and clarify the intended flow, ensuring that all conditions contribute meaningfully to the function's behaviour.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	ex1Staking.sol#L76,77,78,100,119,133,190,234,244,263,267,271 ex1PrivateVesting.sol#L148,149,150,151,152,153,154,155,208,209,210,211,212,213,214,215,258,291,330,355,381,395,403 ex1ICOVesting.sol#L30,86,87,88,89,90,117,118,119,120,121,143,179,213,235,249,257 ex1ICOv2.sol#L19,20,38,39,55,56,57,59,60,123,199,243,255,268,285,301,362,404,439,443,447,451,460,465,470,475 ex1EthICO.sol#L43,44,53,54,55,89,90,91,92,93,123,124,125,126,127,149,163,171,206,211,215
Status	Acknowledged

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.

```
uint256 _percentageReturn
uint256 _timePeriodInSeconds
uint256 _icoStageID
address _caller
Iex1ICO _icoInterface
IVestingICO _vestingInterface
IERC20 _tokenAddress
address _beneficiary
uint256 _totalAmount
uint256 _startTime
uint256 _endTime
uint256 _claimInterval
uint256 _cliffPeriod
uint256 _slicePeriod

...
```

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	ex1Staking.sol#L138,159,163,169,175,197,216,219,224,227 ex1PrivateVesting.sol#L312,317 ex1ICOVesting.sol#L194,198
Status	Acknowledged

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 -
previousStakingRewardClaimTimestamp[_icoStageID][_caller]) *
userRewardPerSecond)

uint256 elapsed = (block.timestamp - cliffTime) / schedule.slicePeriod
uint256 unlocked = (schedule.totalAmount * elapsed) / totalSlices

uint256 unlocked = totalDeposits * elapsedSlices / totalNumberOfSlices
uint256 elapsedSlices = prevClaimTimestamp[_icoStageID][_caller] == 0
            ? (block.timestamp - schedule.startTime) /
schedule.slicePeriod
            : (block.timestamp -
prevClaimTimestamp[_icoStageID][_caller]) / schedule.slicePeriod
```

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**
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

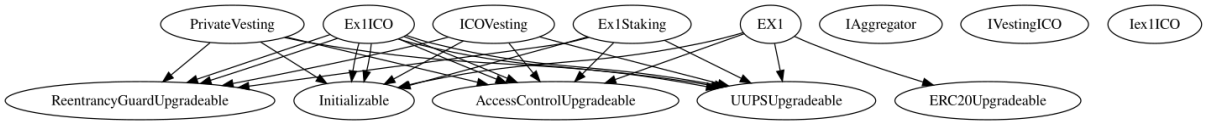
PrivateVesting	Implementation	Initializable, AccessControlUpgradeable, ReentrancyGuardUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	setVestingSchedule	External	✓	onlyRole
	updateVesting	External	✓	onlyValidSchedule onlyRole
	claimTokens	External	✓	nonReentrant onlyValidSchedule 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
	_buyWithUSD	Internal	✓	checkSaleStatus
	purchasedViaEth	External	✓	checkSaleStatus onlyRole
	purchasedViaBTC	External	✓	checkSaleStatus onlyRole
	setMaxTokenLimitPerAddress	External	✓	onlyRole
	setTokenSaleAddress	External	✓	onlyRole
	setMaxTokenLimitPerTransaction	External	✓	onlyRole
	setReceiverWallet	External	✓	onlyRole

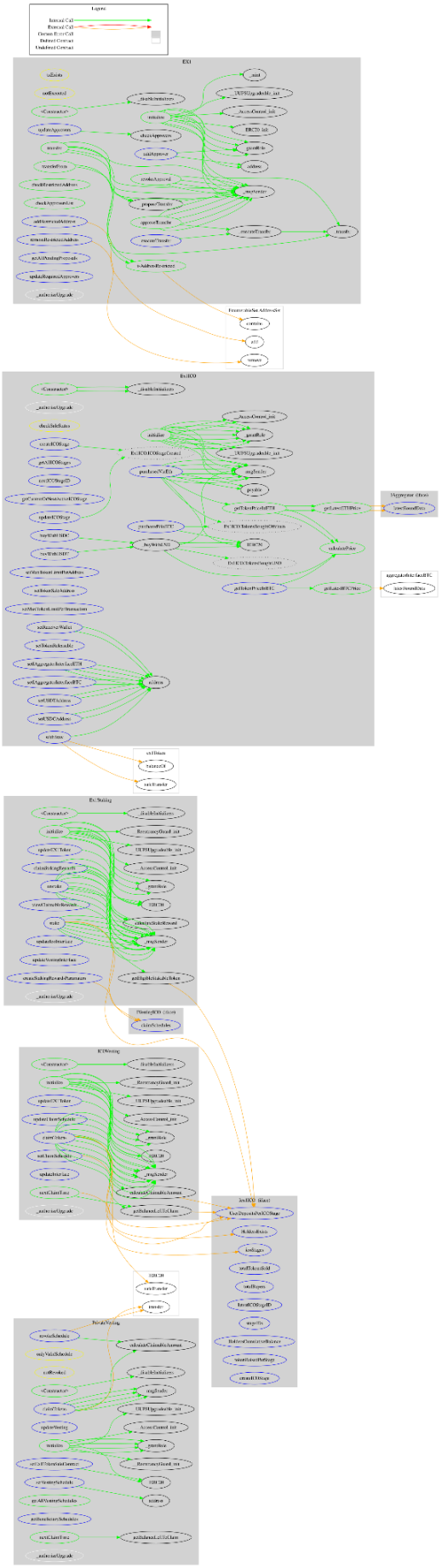
	setTokenReleasable	External	✓	onlyRole
	setIAggregatorInterfaceETH	External	✓	onlyRole
	setIAggregatorInterfaceBTC	External	✓	onlyRole
	setUSDTAddress	External	✓	onlyRole
	setUSDCAddress	External	✓	onlyRole
	withdraw	External	✓	onlyRole
ICOVesting	Implementation	Initializable, AccessControlUpgradeable, 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

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
	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