Cyberscope

Audit Report

eXchange1

September 2025

Repository : https://github.com/ex1ico/ex1SmartContracts

Commits : 8c654f941cba5ea6795b691f32cf989719fc73ab

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	8c654f941cba5ea6795b691f32cf989719fc73ab

Audit Updates

Initial Audit	18 Apr 2025 https://github.com/cyberscope-io/audits/blob/main/ex1/v1/audit.pdf
Corrected Phase 2	13 Jun 2025 https://github.com/cyberscope-io/audits/blob/main/ex1/v2/audit.pdf
Corrected Phase 3	14 Jul 2025 https://github.com/cyberscope-io/audits/blob/main/ex1/v3/audit.pdf
Corrected Phase 4	12 Sep 2025

Source Files

Filename	SHA256
ex1Token.sol	a970f661fad7d32108f5f18695f003dda4545416f66a511035c9f6e907390988
ex1Staking.sol	2aa8d4fd204e678f9b36c012d54a9aa81b47f0d5048ef4f5a2b13a4cd43787a9
ex1PrivateVesting.sol	f53d0ae58f2326408c36f67351ed145b175e2fc5d9e5323e8cc7404dc0a62389

ex1ICOv2.sol	6b5d1a0066ebb5b1beec15083f1e40ffaa645fa549389c39404addca6d091135
ex1ICOVesting.sol	f517925c7615654c9c3a99d6e39e880ba6f2beb33230bd89e6d712bc7b357716

Overview

The suite of five smart contracts, EX1, Ex1ICO , 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 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/ETH, and off-chain 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.

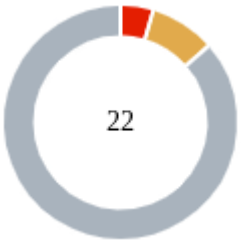
ex1Staking file

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

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	1
Medium	2
Minor / Informative	19

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	2	0	0	0
Minor / Informative	7	12	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	ZCP	Zero Cost Purchase	Unresolved
●	EAR	Excessive Allowance Requirement	Unresolved
●	RCI	Revoke Count Imbalance	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Acknowledged
●	IDU	Inconsistent Data Updates	Acknowledged
●	IVS	Inefficient Vesting Setup	Acknowledged
●	MVN	Misleading Variables Naming	Unresolved
●	MCIC	Missing Claim Interval Check	Unresolved
●	MEE	Missing Events Emission	Acknowledged
●	MTDC	Missing Token Decimal Check	Acknowledged
●	MU	Modifiers Usage	Acknowledged
●	ODM	Oracle Decimal Mismatch	Acknowledged
●	POSD	Potential Oracle Stale Data	Acknowledged

●	PTRP	Potential Transfer Revert Propagation	Acknowledged
●	PUR	Potentially Unclaimed Rewards	Unresolved
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	UIC	Unreachable If Condition	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

ZCP - Zero Cost Purchase

Criticality	Critical
Location	ex1ICOv2.sol#L394
Status	Unresolved

Description

The `_buyWithUSD` method allows users to purchase tokens at predefined prices. The contract relies on calculatePrice to estimate the purchase cost. This estimated amount is then divided by `10**12` and transferred from the user's account to the contract. However, if the calculated cost is less than this factor due to rounding, the user ends up paying 0 tokens, while still receiving the purchased tokens. By repeatedly exploiting this condition, a user can acquire a significant number of tokens for effectively no cost.

```shell
IERC20(_token).safeTransferFrom(_msgSender(),
receivingWallet, usdValue / (10 ** 12));
```

Recommendation

To prevent zero-cost exploitations caused by rounding errors, the team is advised to enforce a minimum purchase amount that aligns with the defined pricing. This ensures that all transactions incur a cost, effectively mitigating the risk of zero-cost attacks.

EAR - Excessive Allowance Requirement

Criticality	Medium
Location	ex1ICOv2.sol#L384
Status	Unresolved

Description

The contract assumes that all price values are denominated with 18 decimal places, interpreting 1 USD as 1e18 in internal calculations. However, this creates a mismatch during the allowance check. The contract compares the USD value, scaled to 18 decimals, directly against the token allowance, without adjusting for the token's actual decimal precision. For example, if the token uses 6 decimals, the contract would require an allowance of 1e18 units to represent 1 USD, which translates to 1e12 tokens, far more than necessary. This misalignment forces users to approve disproportionately large token allowances for standard operations.

```
Shell
require(IERC20(_token).allowance(_msgSender(),
address(this)) >= usdValue, "Insufficient allowance");
```

Recommendation

The team is advised to handle the allowance value in the same manner as it is processed during transfers, by scaling the value according to the appropriate decimal precision. This approach ensures consistency across operations and prevents discrepancies caused by mismatched units.

RCI - Revoke Count Imbalance

Criticality	Medium
Location	ex1Token.sol#L303
Status	Unresolved

Description

The contract defines both `approveTransfer` and `revokeApproval` functions to manage transaction approvals. However, while `revokeApproval` increments the `revokeCount`, the `approveTransfer` function does not decrement it when a user who previously revoked their approval decides to approve again. As a result, if a user first approves a transaction, then revokes it, and later re-approves it, the `revokeCount` remains unchanged. This can lead to inconsistencies in the contract's state, such as a transaction being marked as revoked prematurely, even when the actual number of revocations is below the required threshold.

```shell
function revokeApproval(uint256 _txIndex)
public
onlyRole(APPROVER_ROLE)
txExists(_txIndex)
notExecuted(_txIndex)
notRevoked(_txIndex)
{
require(hasVoted[_txIndex][_msgSender()], "Signer has not
approved this transaction!");

transactions[_txIndex].revokeCount += 1;
if (transactions[_txIndex].approvalCount > 0) {
transactions[_txIndex].approvalCount -= 1;
}

hasVoted[_txIndex][_msgSender()] = false;

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

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

Recommendation

To maintain consistency between approval and revocation states, the contract should ensure that the `revokeCount` is decremented when a user transitions from a revocation back to an approval. This adjustment would keep the approval and revocation counts synchronized and prevent premature or incorrect status changes.

CR - Code Repetition

Criticality	Minor / Informative
Location	ex1Staking.sol#L145,194
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.

```
Shell
function _calculateStakeRewardView(uint256 _icoStageID,
address _caller) internal view returns (uint256) {...}

function calculateStakeReward(uint256 _icoStageID, address
_caller) internal returns (uint256) {...}
```

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
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.

```
Shell
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#L212 ex1ICOVesting.sol#L116 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.

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

    ...
```

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

```
Shell
    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.

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.

```shell
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.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L421,467
Status	Unresolved

Description

Variables can be misleading when their names do not clearly represent the data they store or their intended purpose. In this case, the mappings `BoughtWithEth` and `BoughtWithBTC` suggest they track the amount of tokens purchased with ETH and BTC, respectively. However, the amount of ETH and BTC used for the purchases is stored. This inconsistency can cause confusion and make the code more difficult to interpret and maintain.

```
Shell
BoughtWithEth[_msgSender()] += ethAmount;
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.

MCIC - Missing Claim Interval Check

Criticality	Minor / Informative
Location	ex1PrivateVesting.sol#L147
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.

Shell

```
    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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	ex1Token.sol#L132,156 ex1ICOv2.sol#L488
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.

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

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

```shell
 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;
    }
```

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.

MTDC - Missing Token Decimal Check

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L507,512
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.

```shell
    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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	ex1Token.sol#L287 ex1ICOv2.sol#L143 ex1Staking.sol#L132
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.

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

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

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

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

...
```

```shell
`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	ex1ICOv2.sol#L268
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.

```
Shell
    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.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L243
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.

```
Shell
    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	ex1ICOv2.sol#L425
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.

```
Shell
(bool successful,) =
payable(receivingWallet).call{value: ethAmount}("");
require(successful, "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.

PUR - Potentially Unclaimed Rewards

Criticality	Minor / Informative
Location	ex1Staking.sol#L199
Status	Unresolved

Description

The `stake` method distributes rewards based on the duration each user has staked. The `userRewardPerSecond` is calculated proportionally to the user's deposit and the full `timePeriodInSeconds` of the staking stage. This approach may lead to inconsistencies in reward distribution, as users who stake for shorter durations still have their rewards spread across the entire staking period. Consequently, users who stake late in the stage may only be able to claim a small portion of their expected rewards.

```shell
uint256 userRewardPerSecond = userPercentage /
stakingParameters[_icoStageID].timePeriodInSeconds;
```

Recommendation

The team is advised to validate the implementation logic of the reward distribution to ensure it aligns with the intended design. Alternatively, the team may consider distributing rewards proportionally based on the remaining time period.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	ex1ICOv2.sol#L346 ex1ICOVesting.sol#L171 ex1Staking.sol#L135,318 ex1Token.sol#L343
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.

```
Shell
bool success =
IERC20(ex1Token).transfer(_recipient, _amount);
require(
    success,
    "Token Transfer Failed!"
);
```

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

```shell
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#L180 ex1PrivateVesting.sol#L291
Status	Acknowledged

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.

```
Shell
 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];
    }
```

Shell

```
    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	eX1token.sol#L103,132,156,172,180,189,218,228,244,280,303,343,378 ex1Staking.sol#L54,76,77,78,101,131,194,252,296,306,325,329,333 ex1PrivateVesting.sol#L148,149,150,151,152,153,154,155,209,210,211,212,213,214,215,216,259,292,331,356,382,396,404 ex1ICOVesting.sol#L30,63,86,87,88,89,90,117,118,119,120,121,144,180,214,236,250,258 ex1ICOv2.sol#L22,23,41,42,58,59,60,62,63,136,212,268,280,293,312,329,406,448,476,480,484,488,497,502,507,512
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.

```shell
uint256 _required
address[] memory _approver
address _address
uint256 _value
address _to
address _from
uint256 _txIndex
address _admin
address _vestingAddress
address _icoAddress
address _tokenAddress
uint256 _percentageReturn
uint256 _timePeriodInSeconds
uint256 _icoStageID

...
```

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	eX1token.sol#L229,247
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.

```shell
isAddressRestricted(_msgSender()) == true
isAddressRestricted(_from) == 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#L152,171,174,179,182,199,220,224,230,236,259,278,281,286,289 ex1PrivateVesting.sol#L313,318 ex1ICOVesting.sol#L195,199
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.

```shell
uint256 userRewardPerSecond = userPercentage /
stakingParameters[_icoStageID].timePeriodInSeconds
reward = ((time - stakeTimestamp[_icoStageID][_caller]) *
userRewardPerSecond)
uint256 elapsed = (block.timestamp - schedule.startTime) /
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	✓	nonReentrant
	claimStakingRewards	External	✓	nonReentrant
	_calculateStakeRewardView	Internal		
	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 checkAddressNotRestricted
	buyWithUSDT	External	✓	checkSaleStatus checkAddressNotRestricted
	_processTokenAllocation	Internal	✓	
	_buyWithUSD	Internal	✓	checkSaleStatus checkAddressNotRestricted
	purchasedViaEth	External	Payable	nonReentrant checkSaleStatus checkAddressNotRestricted

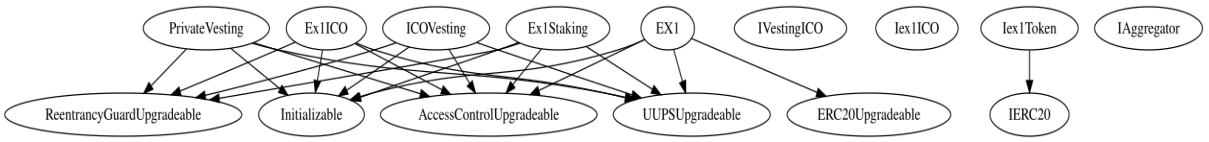
	purchasedViaBTC	External	✓	checkSaleStatus checkAddressNotRestricted onlyRole
	setMaxTokenLimitPerAddress	External	✓	onlyRole
	setTokenSaleAddress	External	✓	onlyRole
	setMaxTokenLimitPerTransaction	External	✓	onlyRole
	setReceiverWallet	External	✓	onlyRole
	setTokenReleasable	External	✓	onlyRole
	setIAggregatorInterfaceETH	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
EX1	Implementation	Initializable, ERC20Upgradeable, AccessControlUpgradeable, UUPSUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	removeApprovers	External	✓	onlyRole
	addApprover	External	✓	onlyRole
	addRestrictedAddress	External	✓	onlyRole
	removeRestrictedAddress	External	✓	onlyRole
	isAddressRestricted	Public		-
	checkRestrictedAddress	Public		-
	checkApproversList	Public		-
	checkApprovers	Public		-
	transfer	Public	✓	-
	transferFrom	Public	✓	-
	_proposeTransfer	Internal	✓	

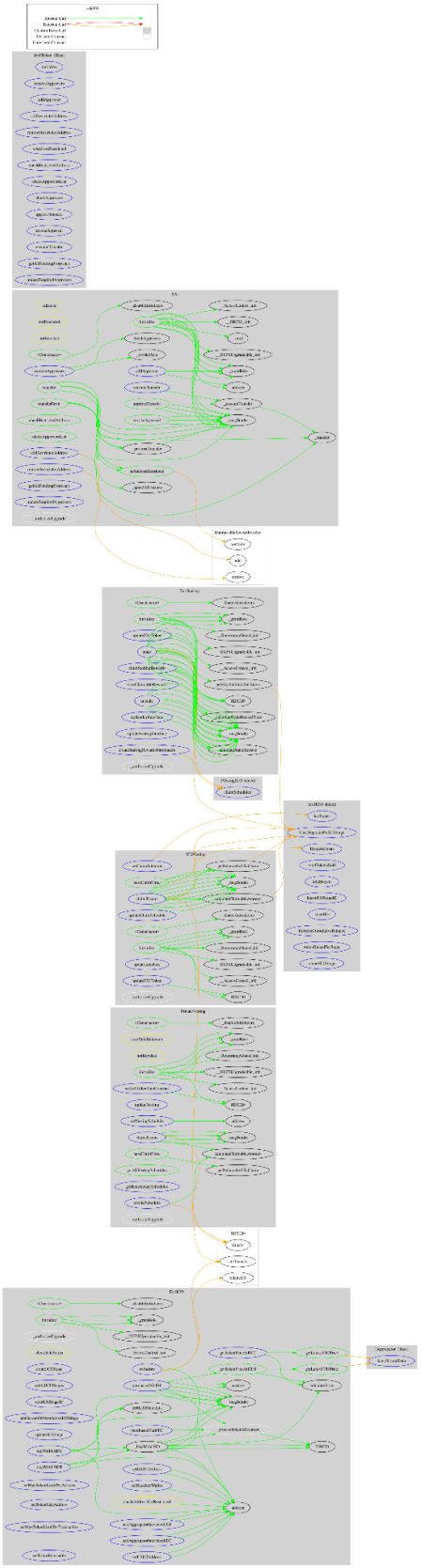
	approveTransfer	Public	✓	onlyRole txExists notExecuted notRevoked
	revokeApproval	Public	✓	onlyRole txExists notExecuted notRevoked
	_executeTransfer	Internal	✓	txExists notExecuted
	executeTransfer	External	✓	txExists notExecuted onlyRole
	getAllPendingProposals	External		-
	updateRequiredApprovers	External	✓	onlyRole
	_authorizeUpgrade	Internal	✓	onlyRole
IVestingICO	Interface			
	claimSchedules	External	✓	-
Iex1ICO	Interface			
	totalTokensSold	External		-
	totalBuyers	External		-
	latestICOStageID	External		-
	stageIDs	External		-
	HoldersCumulativeBalance	External		-
	tokenRaisedPerStage	External	✓	-
	UserDepositsPerICOStage	External		-
	HoldersExists	External		-
	icoStages	External		-

	createICOStage	External	✓	-
Iex1Token	Interface	IERC20		
	initialize	External	✓	-
	removeApprovers	External	✓	-
	addApprover	External	✓	-
	addRestrictedAddress	External	✓	-
	removeRestrictedAddress	External	✓	-
	isAddressRestricted	External		-
	checkRestrictedAddress	External		-
	checkApproversList	External		-
	checkApprovers	External		-
	approveTransfer	External	✓	-
	revokeApproval	External	✓	-
	executeTransfer	External	✓	-
	getAllPendingProposals	External		-
	updateRequiredApprovers	External	✓	-
IAggregator	Interface			
	latestRoundData	External		-

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, 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