hexens x TRENDEX.ONE

Security Review Report for TrenDex.One

January 2026

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for TrenDex.One, a token deployment protocol. This review included all of the core EVM code.

Our security assessment was a full review of the code, spanning a total of 1.5 weeks.

During our review, we did not identify any major security vulnerability.

We did identify some minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

Review Led by

Trung Dinh, Lead Security Researcher

Scope

The analyzed resources are located on:

tx-smart-contracts@6ff629113f5c4d5ad333dd69f95fbaf21b67fc25

The issues described in this report were fixed in the following commit:

tx-smart-contracts@37a5e5c5f0b77ad0aac45c9e3991205498079383

Changelog

29 December 2025	Audit start
7 January 2026	Initial report
12 January 2026	Revision received
13 January 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

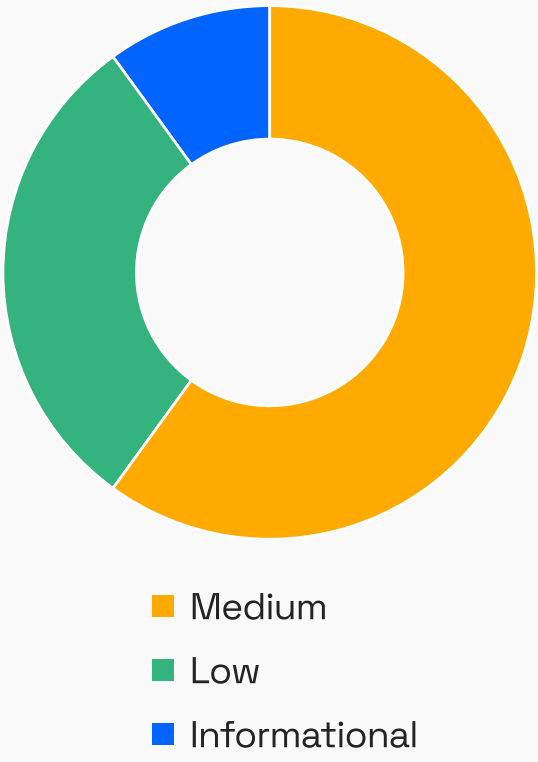
- **Issue Symbolic Codes**

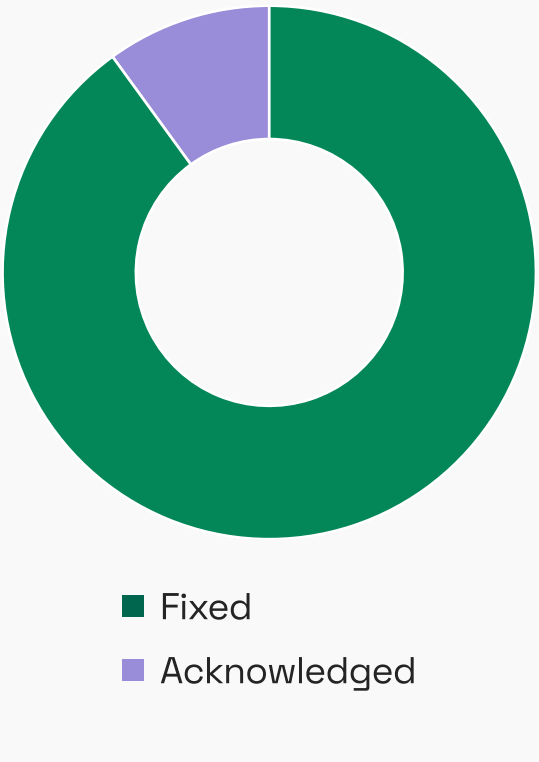
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	6
Low	3
Informational	1
Total:	**10**





6. Weaknesses

This section contains the list of discovered weaknesses.

TRDX-9 | PoolContract is vulnerable to a first deposit attack

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/core/PoolContract.sol#L231-L237

Description:

The **PoolContract.addFunds()** function is used to add USDC liquidity to the pool. This function is not restricted to any actors, which allows anyone to increase the value of the token price by transferring USDC to the contract.

This creates a flaw where the first depositor can manipulate the token price and, in the worst case, steal the next user's deposited funds.

Consider the following example:

1. We assume that there is no presale period and no fee, and the initial **presaleTokenPrice** is **1e18**.
2. Alice (the attacker) triggers buyTokens() with 1 wei of USDC.
   - **usdcBalance = 0 + 1 = 1**
   - **totalSoldTokens = 0 + 1 = 1**
3. Bob (the victim) decides to buy tokens with 1000 USDC (**buyTokens(1000 * 1e6)**), expecting to receive **1000 * 1e6** shares in return.
4. Alice sees Bob's transaction and decides to front-run it by increasing the USDC liquidity by 500 USDC by triggering **addFunds(500 * 1e6)**.
   - **usdcBalance = 1 + 500 * 1e6**
   - **getTokenPrice() = (1 + 500 * 1e6) * 1e18 / 1 = (1 + 500 * 1e6) * 1e18**
5. Bob's buy token transaction is executed. He receives:
   - **getTokensByUsdc(1000 * 1e6) = (1000 * 1e6) * 1e18 / ((1 + 500 * 1e6) * 1e18) = 1** token
   - **usdcBalance = 1 + 500 * 1e6 + 1000 * 1e6 = 1 + 1500 * 1e6**
   - totalSoldTokens = 1 + 1 = 2
6. Alice calls **sellTokens(1)** to sell all of her tokens. She receives:
   - **getUsdcByTokens(1) = 1 * ((1 + 1500 * 1e6) * 1e18 / 2) / 1e18 = 750 * 1e6** wei of USDC

As we can see, Alice makes 750 USDC at the cost of 500 (+1 wei) USDC.

Remediation:

To remediate the issue, we recommend implementing a slippage check in the **buyTokens()** and **sellTokens()** functions. This check ensures that the actual amount received matches the user's expected amount. (This is because the token price can fluctuate even when it is not due to an attacker's actions.)

```
-- function buyTokens(uint256 _usdcAmount) external whenNotPaused {
++ function buyTokens(uint256 _usdcAmount, uint256 _minTokenAmount) external whenNotPaused {
    ...

    if (tokensToBuy == 0) revert InsufficientUSDC();
++    if (tokensToBuy < _minTokenAmount) revert InsufficientTokens;

    ...
  }

-- function sellTokens(uint256 _tokenAmount) external whenNotPaused {
++ function sellTokens(uint256 _tokenAmount, uint256 _minUsdcAmount) external whenNotPaused {
    if (usdcBalance < userAmount + feeAmount) revert InsufficientUSDC();
++    if (usdcBalance < _minUsdcAmount) revert InsufficientUSDC();

    ...
  }
```

To fully mitigate the donation attack, note that the formulas used to calculate the amount of tokens to buy and the amount of USDC received when selling closely mirror the ERC-4626 vault model, where USDC represents the assets and the token represents the shares. As a result, it is advisable to adopt the virtual shares and virtual assets approach proposed in OpenZeppelin's ERC-4626 design, as discussed here:
ERC4626 inflation attack mitigation by Amxx · Pull Request #3979 · OpenZeppelin/openzeppelin-contracts

Alternatively, the protocol can perform an initial **buyTokens()** operation to prevent the issue.

Furthermore, we should implement the similar fix to the contract **StakingTrenDex** either.

TRDX-3 | Dust Remainders in Share Conversion Can Corrupt Total Asset Accounting

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/core/StakingTrenDex.sol

Description:

The current design of the **StakingTrenDex** contract combines the staked **TrenDexOne (TxOne)** and accumulated **USDC** rewards into a single notion of total assets, treating both tokens as having the same value.

```
function totalAssetsValue() public view returns (uint256) {
    return totalStakedAssets + usdcToAsset(totalRewardsInUSDC);
}
```

Under this model, any portion of the assets attributable to a user's shares that exceeds the user's recorded stake (**userStakedAssets[user]**) is treated as USDC rewards.

```
uint256 _userStaked = userStakedAssets[_user];
if (_userStaked == 0 || _assetsToWithdraw > _userStaked) {
    return (0, 0);
}

uint256 _userShares = balanceOf(_user);
uint256 _totalAssets = convertToAssets(_userShares);

uint256 _currentRewardAssets = _totalAssets > _userStaked
    ? _totalAssets - _userStaked
    : 0;
uint256 _currentRewardUSDC = assetToUsdc(_currentRewardAssets);
```

However, an issue arises when not all excess assets are necessarily USDC. A user's **TrenDex.One** stake can also increase due to dust remainders created when converting assets to shares.

```
function convertToShares(uint256 _assets) public view returns (uint256) {
    uint256 supply = totalSupply();
    uint256 totalAssets = totalAssetsValue();

    if (supply == 0 || totalAssets == 0) {
        return _assets;
    }

    return _assets.mulDiv(supply, totalAssets, Math.Rounding.Floor);
}
```

The **convertToShares()** function performs a round-down calculation. Any remainder from the division of **_assets * supply** by **totalAssets** is discarded and not accounted for. Over time, this dust implicitly inflates **totalAssetsValue()**, which breaks the intended accounting between **USDC** rewards and **TxOne** stake.

Consider the following example:

1. Initial state
   Assume the exchange rate between shares and assets is 1:2 (Bob has 1 share corresponding to 2 TxOne):
   - **totalSupply() = 1**
   - **totalStakedAssets = 2**
   - **userStakedAssets[Bob] = 2**
2. Alice deposits 3 **TxOne**
   Alice receives 1 share; the remaining dust of 1 asset is not used to mint any share:
   - **totalSupply() = 2**
   - **totalStakedAssets = 5**
   - Rate → 2:5
3. Alice deposits 4 **TxOne**
   Alice receives 1 additional share; the remaining dust of 2 assets is not used to mint any share:
   - **totalSupply() = 3**
   - **totalStakedAssets = 9**
   - Rate → 3:9 = 1:3

As a result, Bob's 1 share can now redeem 3 assets. The contract incorrectly treats the extra 1 asset (3 − 2 = 1) as USDC rewards, even though no USDC rewards exist.

This ultimately locks Bob's stake, as the contract attempts to transfer non-existent USDC rewards to him.

Remediation:

Consider taking every remainder into account.

For example, in the **deposit()** function, the amount of deposited TxOne should be required to satisfy that **_assets * supply** is divisible by **totalAssets**.

TRDX-1 | Missing Slippage Protection Allows Sandwich Attack

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/core/GlobalPool.sol#L357-L367
contracts/core/GlobalPool.sol#L331-L340

Description:

When swapping through the Balancer Router, the **minAmountOut** parameter is set to **0**, providing no slippage protection. This allows an attacker to perform a sandwich attack and manipulate the swap price.

```
_amountOut = balancerRouter.swapSingleTokenExactIn{value: 0}(
    balancerPool,
    usdc,
    trenDexOne,
    _amountIn,
    0,  // @audit minAmountOut = 0 - no slippage protection
    block.timestamp + 300,
    false,
    bytes("0x0")
);
```

Attack Scenario:

1. An attacker detects a **GlobalPool.addFunds()** transaction in the mempool
2. The attacker front-runs by buying a large amount of TxOne tokens, pushing the price up
3. The GlobalPool swap executes at the manipulated (inflated) price
4. The attacker back-runs by selling TxOne tokens to extract profit

Remediation:

Consider using an appropriate **minAmountOut** value to ensure proper slippage protection.

TRDX-5 | Reward Sandwiching in StakingTrenDexOne

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/core/StakingTrenDex.sol#L249-L260

Description:

Rewards are added to the **StakingTrenDex** contract by an account with the **MANAGER_ROLE** via the **sendRewards()** function. When rewards are transferred into the contract, **totalAssetsValue()** increases while **totalSupply()** remains unchanged, thereby improving the exchange rate for all existing shareholders.

A malicious actor can exploit this behavior by depositing TrenDexOne immediately before **sendRewards()** is called, allowing them to instantly capture a portion of the newly added rewards that were intended for long-term stakers. The attacker can then promptly claim the rewards and unstake, recovering their TrenDexOne while extracting the rewards.

```
/// @notice Add USDC rewards to the vault (manager only)
/// @param _amountUSDC Amount of USDC to add to the reward pool
function sendRewards(
  uint256 _amountUSDC
) external onlyRole(MANAGER_ROLE) whenNotPaused {
  if (_amountUSDC == 0) revert InvalidAmount();

  totalRewardsInUSDC += _amountUSDC;
  rewardToken.safeTransferFrom(msg.sender, address(this), _amountUSDC);

  emit RewardsAdded(msg.sender, _amountUSDC);
}
```

Remediation:

Consider implementing a staking/unstaking fee or a “warmup period” that requires depositors to wait for a certain period of time before withdrawing after a deposit.

TRDX-8 | Unable to Finalize Prediction Event When Global Share Is Zero

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/core/PredictionContract.sol#L598-L599

Description:

The function **PredictionContract._distributeShares()** is responsible for distributing shares to the platform, the local pool, and the global pool after a prediction event is resolved.

```
function _distributeShares(
    uint256 _predictionId,
    PredictionEvent storage _eventData
) internal {
    uint256 totalPot = _eventData.totalPot;

    ...

    uint256 globalShare = (totalPot * _eventData.globalPoolPercentage) /
        PERCENTAGE_DENOMINATOR;

    ...

    // Global pool share
    globalPool.addFunds(globalShare); // @audit globalShare could be 0

    ...
}
```

The calculated **globalShare** is forwarded to the global pool via **globalPool.addFunds(globalShare)**. However, there is no check to ensure that **globalShare** is greater than zero before invoking this function.

If **globalPoolPercentage** is set to **0**, the computed **globalShare** will also be **0**. As a result, calling GlobalPool.addFunds(0) will revert, as shown below:

```
function addFunds(uint256 _amount) external nonReentrant {
    if (_amount == 0) revert ZeroAmount(); // @audit revert here

    ...
}
```

This causes the entire distribution process to revert, preventing the prediction event from being finalized and blocking the distribution of funds to all parties.

Remediation:

Add a conditional check before calling **globalPool.addFunds()** to ensure the amount is greater than zero, or modify **addFunds()** to gracefully handle zero-value transfers when appropriate.

TRDX-11 | Incorrect Round-Down Share Burning Allows Excess USDC Extraction

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/core/StakingTrenDex.sol#L380-L384

Description:

The function **StakingTrenDex.withdrawStaked()** allows users to withdraw a portion of their staked TxOne along with the corresponding USDC rewards. Internally, it relies on **previewWithdrawStaked()** to calculate the number of **shares** that should be burned for the withdrawing user.

```
function previewWithdrawStaked(
    uint256 _assetsToWithdraw,
    address _user
) public view returns (uint256 sharesToBurn, uint256 rewardUSDC) {
    ...

    sharesToBurn = _userShares.mulDiv(
        _assetsToWithdraw,
        _userStaked,
        Math.Rounding.Floor
    );
    rewardUSDC = _currentRewardUSDC.mulDiv(
        _assetsToWithdraw,
        _userStaked,
        Math.Rounding.Floor
    );

    ...
}
```

From the implementation, we can observe that **previewWithdrawStaked()** computes **sharesToBurn** using round-down division. This is incorrect because, during withdrawals, the number of shares to burn should be rounded in favor of the protocol. In this case, a round-up calculation should be used instead.

Failing to do so enables an attack vector where a user can withdraw a sufficiently small **_assetsToWithdraw** such that:

```
_userShares * _assetsToWithdraw < _userStaked
```

As a result, **sharesToBurn** becomes **0**, allowing the user to withdraw assets and rewards without burning any shares. In the worst case, this can be exploited to extract excess USDC rewards from the protocol.

Consider the example scenario

1. Initial state
   - Assume there are two suppliers with the following balances (note that this state of share rate is rare):
     - Alice
       - **balanceOf(Alice) = 2**
       - **userStakedAssets[Alice] = 200 × 10^18**
     - Bob
       - **balanceOf(Bob) = 2**
       - **userStakedAssets[Bob] = 200 × 10^18**
     - **USDC.balanceOf(StakingTrenDex) = 0**
2. USDC rewards distribution
   - **200 × 10^6** USDC rewards are sent to the contract.
   - Alice and Bob are each entitled to **100 × 10^6** USDC.

```
totalAssetsValue() = 400 × 10^18 + (200 × 10^6 × 10^12)
                   = 600 × 10^18
```

3. Alice withdraws part of her stake
   - Alice calls **withdraw(50 × 10^18)**.
     - Shares burned:

```
sharesToBurn = 2 × (50 × 10^18) / (200 × 10^18)
             = 0
```

     - USDC rewards withdrawn:

```
rewardUSDC = (100 × 10^6) × (50 × 10^18) / (200 × 10^18)
           = 25 × 10^6
```

- State after withdrawal:
  - **balanceOf(Alice) = 2**
  - **userStakedAssets[Alice] = 150 × 10^18**

```
totalAssetsValue() = 150 × 10^18 + 200 × 10^18 + (175 × 10^6 × 10^12)
                   = 525 × 10^18
```

4. Alice unstakes completely

- Alice then calls **unstake()**.
  - Total assets allocated to Alice:

```
_currentTotalAssets = (525 × 10^18) × 2 / 4
                    = 262 × 10^18
```

  - Reward assets:

```
_totalRewardAssets = 262 × 10^18 − 150 × 10^18
                   = 112 × 10^18
```

  - Reward in USDC:

```
_totalRewardUSDC = 112 × 10^18 / 10^12
                 = 112 × 10^6
```

As a consequence, Alice is able to claim a total of:

```
(112 + 25) × 10^6 = 137 × 10^6 USDC
```

This results in an excess profit of **37 × 10^6** USDC, directly attributable to the incorrect round-down behavior when calculating **sharesToBurn**.

```
sharesToBurn = _userShares.mulDiv(
    _assetsToWithdraw,
    _userStaked,
    Math.Rounding.Floor
);
```

Remediation:

To prevent users from withdrawing assets and rewards without burning the appropriate number of shares, the calculation of **sharesToBurn** in **previewWithdrawStaked()** must be rounded up.

TRDX-2 | Refund Batch Processing Can Be Permanently Blocked by Blacklisted Address

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/core/PredictionContract.sol#L346-L352

Description:

The function **PredictionContract.cancelPredictionEventAndRefund()** is used to cancel a prediction event and refund all users in batches.

In each batch, the function iterates over predictions (up to a maximum of 100) and transfers **1 USDC** back to the prediction's owner. This design introduces a risk: if a user becomes "malicious" and is blacklisted by USDC, the transfer to that address will revert, causing the entire refund batch to fail.

As a consequence, the refund process can be permanently blocked, effectively freezing all funds associated with the prediction event.

```
for (uint256 i = cursor; i < end; ++i) {
    address user = predictions[i].user;
    if (user != address(0)) {
        usdc.safeTransfer(user, PREDICTION_PRICE);  // @audit reverts on failure
        refundedCount++;
    }
}
```

Remediation:

Consider replacing the push-based refund mechanism with a pull-based refund model. Instead of transferring USDC directly during batch processing, record each user's refundable amount in storage and allow users to claim their refunds individually.

```
for (uint256 i = cursor; i < end; ++i) {
    address user = predictions[i].user;
    if (user != address(0)) {
--      usdc.safeTransfer(user, PREDICTION_PRICE);
++      pendingRefunds[user] += PREDICTION_PRICE;
        refundedCount++;
    }
}
```

TRDX-7 | Users Can Submit Predictions During/After Event Cancellation or Resolve a Canceled Event

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

contracts/core/PredictionContract.sol

Description:

When a prediction event is canceled via **cancelPredictionEventAndRefund** and refunds are in progress, users can still submit new predictions. It is also possible to call **resolveWithMerkleRoot** after the first refund batch, which finalizes the event even though it is in the process of being canceled.

For example, if 200 predictions were submitted and the first refund batch processes 100 of them, a call to **resolveWithMerkleRoot** would include only the remaining 100 predictions, even though the event has been canceled.

Remediation:

Consider introducing an **enum** to track the event status:

```
enum EventStatus { Inactive, CancelInProgress, Resolved, Active }
```

Add a **status** field to **PredictionEvent** and update functions to enforce the event to have the correct state for the action.

```
if (eventData.status != EventStatus.Active) revert
```

TRDX-10 | Incorrect Timestamp Validation in editPredictionEvent

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/core/PredictionContract.sol#L297

Description:

The function **PredictionContract.editPredictionEvent()** allows updating the start and end times of an unresolved prediction event.

```
function editPredictionEvent(
    uint256 _predictionId,
    uint64 _newStartTime,
    uint64 _newEndTime
) external onlyRole(CREATOR_ROLE) {
    PredictionEvent storage eventData = predictionEvents[_predictionId];
    if (eventData.resolved) revert AlreadyResolved();
    if (_newStartTime >= _newEndTime) revert InvalidStartTime();
    if (
        _newEndTime <= block.timestamp ||
        _newEndTime >= eventData.resolutionTime
    ) revert InvalidEndTime();
    if (
        eventData.startTime > block.timestamp &&
        _newStartTime > block.timestamp
    ) {
        eventData.startTime = _newStartTime;
    }
    eventData.endTime = _newEndTime;

    emit PredictionEventEdited(_predictionId, _newStartTime, _newEndTime);
}
```

The validation at line 297 is intended to ensure that the new start time precedes the new end time. However, this check is insufficient because **_newStartTime** is only applied if the conditions at lines 303–304 are met.

Specifically, when **eventData.startTime <= block.timestamp** or **_newStartTime <= block.timestamp**, the event's **startTime** remains unchanged. In such cases, it is possible for the updated **endTime** to become earlier than the effective startTime, resulting in an invalid event configuration.

Remediation:

Consider modifying the code as follows:

```
function editPredictionEvent(
    uint256 _predictionId,
    uint64 _newStartTime,
    uint64 _newEndTime
) external onlyRole(CREATOR_ROLE) {
    PredictionEvent storage eventData = predictionEvents[_predictionId];
    if (eventData.resolved) revert AlreadyResolved();
-   if (_newStartTime >= _newEndTime) revert InvalidStartTime();
    if (
        _newEndTime <= block.timestamp ||
        _newEndTime >= eventData.resolutionTime
    ) revert InvalidEndTime();
    if (
        eventData.startTime > block.timestamp &&
        _newStartTime > block.timestamp
    ) {
        eventData.startTime = _newStartTime;
    }
    eventData.endTime = _newEndTime;

+   if (eventData.startTime >= eventData.endTime) revert InvalidStartTime();

    emit PredictionEventEdited(_predictionId, _newStartTime, _newEndTime);
}
```

TRDX-4 | Typo in the codebase

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

There are several typographical errors in the codebase:

1. contracts/core/GlobalPool.sol#L163
   The function name contains a typo: **editPremit2Approval** should be **editPermit2Approval**.

```
function editPremit2Approval(
```

2. contracts/core/GlobalPool.sol#L287C16–L287C25
   The comment contains a typo: **wWithdrawal** should be **Withdrawal**.

```
* @notice wWithdrawal — only owner can rescue any ERC20
```

Remediation:

Correct the identified typographical errors to improve code clarity and maintainability.

hexens x TRENDEX.ONE