hexens x t3rn

Security Review Report for t3rn

June 2025

Table of Contents

- Price Calculation Ignores Token Decimals in OpenMarketPricer
- Inverted Time-Factor Causes Rewards to Decrease Near Week End
- Order can be confirmed multiple times
- Unused events
- BRN contract constructor ignores owner parameter
- Missing Zero-Check in setRatio May Lead to Division by Zero
- Duplicate call value amount check
- Inconsistent Action-Type IDs Between AttestationsVerifierProofs and ClaimerGMPV2

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 audit of t3rn, a universal execution protocol. It allows for messaging and asset movement across ecosystems.

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

During our audits, we have identified three critical severity vulnerabilities, which could have lead to loss of principal assets.

We have also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged 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

Kasper Zwijsen, Head of Audits

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/t3rn/l2-settlements-contracts/tree/85a449cf59c6f8d88fc94595376d3f7e7a64dc61

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

🔗 https://github.com/t3rn/l2-settlements-contracts

📌 Commit: `0fe52417dee41d38d9dc2279e65fce2b02474a63`

▪ Changelog

11 June 2025	Audit start
24 June 2025	Initial report
26 June 2025	Revision received
03 July 2025	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.

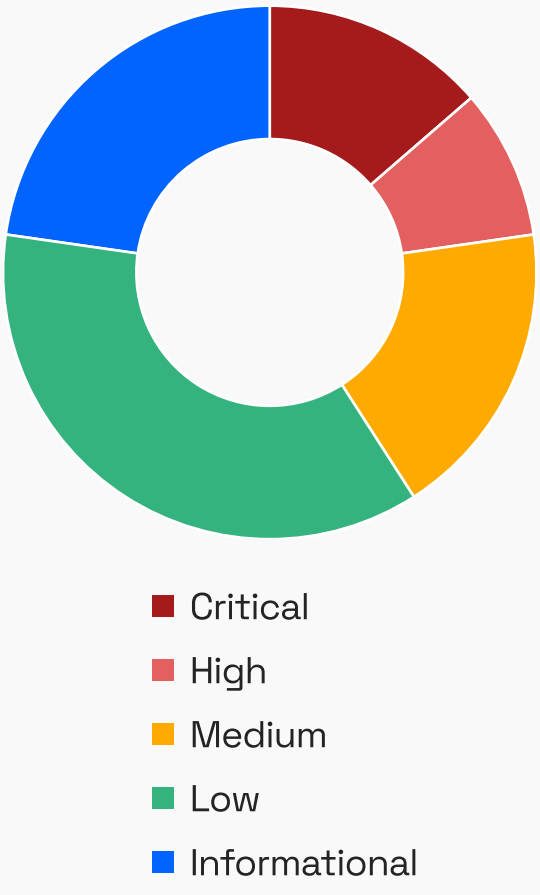
- **Issue Symbolic Codes**

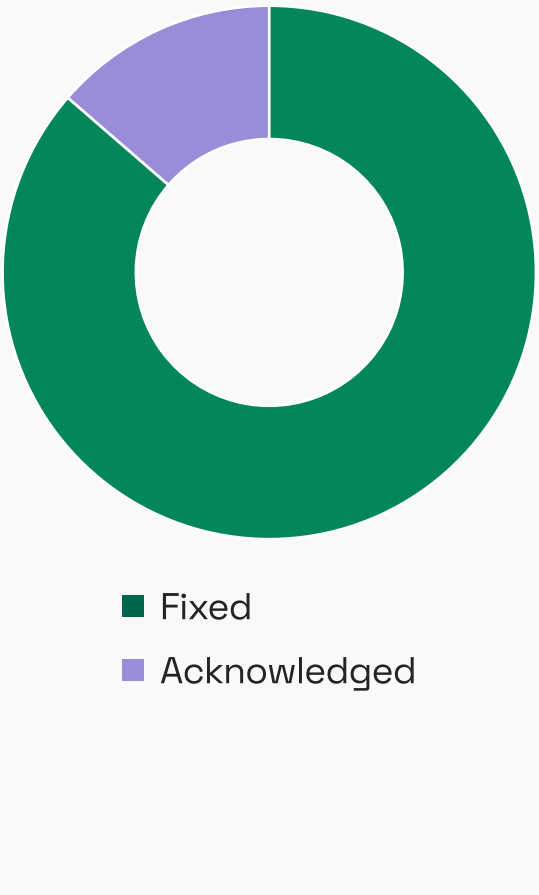
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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

5. Findings Summary

Severity	Number of findings
Critical	3
High	2
Medium	3
Low	9
Informational	5
Total:	**22**





6. Weaknesses

This section contains the list of discovered weaknesses.

T3RN1-4 | Refund claiming mechanism can be used to steal funds

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	Critical

Path:

contracts/remoteOrder.sol#L395-407
contracts/claimerGMPV2.sol#L199-206

Description:

In the **remoteOrder** contract when the user creates an order they can later refund the order to receive the assets back. The refund is done by calling the **claimRefundV2** function which has the following check:

```
require(
    claimerGMP.checkIsRefundable(
        orderId,
        orderTimestamp,
        executionCutOff,
        rewardAsset,
        maxReward,
        msg.sender,
        _batchPayloadHash,
        _batchPayload
    ),
    "RO#1"
);
```

The function in turn has the following checks:

```
if (checkIsRefundableWithEscrow(orderId, orderTimestamp, orderTimeout,
rewardAsset, maxReward, beneficiary)) {
    return true;
}
if (attesters.skipEscrowWrites() || forceCheckV2) {
    uint8 actionType = 1;
    return isClaimable(_batchPayloadHash, _batchPayload, orderId, beneficiary,
maxReward, actionType);
}
return false;
```

The issue stems from the fact that the **rewardAsset** initially given in the order can be different than the one being refunded in the **claimRefundV2** function which would allow an attacker to create an order with lower cost token then refund the order with a higher valuation token because of the following incorrect checks:

- The **checkIsRefundableWithEscrow** will return false because the reward asset is being checked in the last hash check: **return paymentHash == calculatedWithdrawHash;** which works correctly and takes us into the second check
- The **isClaimable** function at no point verifies that the **rewardAsset** in the original order matches the one given in the refund function

Thus the attacker creates an order with low valuation reward asset, transfers a lot of it to the contract, then refunds but instead of getting the low valuation asset back they can steal any asset that's currently on the contract. The worst case scenario would be low valuation but high decimal count token against high valuation low decimal count token.

Remediation:

Add **rewardAsset** check in the **isClaimable** function.

T3RN1-14 | Invalid attester duplicate signature removal leads to quorum bypass via merkle proof forgery

Fixed

Severity:	Probability:	Impact:
Critical	Likely	Critical

Path:

contracts/attestationsVerifierProofs.sol#L252-277
contracts/attestationsVerifierProofs.sol#L311-324

Description:

In the **AttestationsVerifierProofs** contract the **receiveAttestationBatch** function is used to receive payloads from the attesters and to forward that data to the protocol. To verification of whitelisted attesters is being done by supplying the list of the attesters that signed the payload as leaves to the merkle tree multi proof check:

```
if (batch.maybeNextCommittee.length > 0) {
    // Current committee has signed on the next committee - if the batch
proposes committee rotation - verify the signatures against the next committee
hash
    require(
        MerkleProof.multiProofVerifyCalldata(
            multiProofProof,
            multiProofMembershipFlags,
            nextCommitteeHash,
            attestersAsLeaves
        ),
        "PROOF_NEXT_VERIFICATION_FAILED"
    );
    // Update the current committee hash to the next committee hash, since the
batch has been applied meaning new committee has been formed
    bytes32 impliedNextCommitteeHash =
implyCommitteeRoot(batch.maybeNextCommittee);
    currentCommitteeHash = nextCommitteeHash;
    nextCommitteeHash = impliedNextCommitteeHash;
} else {
    require(
        MerkleProof.multiProofVerifyCalldata(
            multiProofProof,
            multiProofMembershipFlags,
            currentCommitteeHash,
```

```
            attestersAsLeaves
        ),
        "PROOF_VERIFICATION_FAILED"
    );
}
```

The if case handles the case where there are new attesters that are needed to be added to the attesters whitelist and the else case handles the case where there are no new attesters. The function also contains a quorum check wherein if the amount of attesters that signed the batch surpasses the quorum the payload is considered valid:

contracts/attestationsVerifierProofs.sol#L247

```
require(attestersAsLeaves.length >= quorum, "INSUFFICIENT_QUORUM");
```

The list of attesters is created by recovering the addresses from signatures using the **recoverCurrentSigners** function:

contracts/attestationsVerifierProofs.sol#L245

```
bytes32[] memory attestersAsLeaves = recoverCurrentSigners(batchMessageHash,
signatures, batch.bannedCommittee);
```

The function goes over the list of signatures, tries to recover the address of the attester and tries to stop a potential malicious attester from bypassing the quorum by skipping duplicate attesters. However the **recoverCurrentSigners** function by itself doesn't have any checks to verify that the address is indeed an attester, its only purpose is to create an attester list without any duplicates and check for banned attesters and return that list to the caller function. The duplicate removal is being done in the following way:

```
bytes32 candidate =
keccak256(bytes.concat(keccak256(abi.encode(recoveredSigner))));
// Dedup leaves by checking if the candidate is already in the leaves array
bool isDuplicate = false;
for (uint256 j = 0; j < i; ++j) {
    if (leaves[j] == candidate) {
        isDuplicate = true;
        break;
    }
}
if (isDuplicate) {
```

```
            continue; // Skip adding duplicate leaves
        }
        uniqueCount++;
        leaves[i] = candidate;
```

The duplicates are not removed and instead are skipped and when a non duplicate valid address is found it's stored in the **leaves[i]** which means that if there were a couple of duplicates before a valid non duplicate one, the array will have a valid address in the array, then a bunch of zeroes, then once again a valid address which would look something like this: **[0x1234,0,0,0,....,0x2345]**.

The array is later trimmed and the length is set to uniqueCount:

contracts/attestationsVerifierProofs.sol#L326-330

```
    // Trim the leaves array to the actual number of unique leaves
    assembly {
        mstore(leaves, uniqueCount)
    }
    return leaves;
```

However in this case the array can have a look like this:
[0x1234,0,0,0,0x2345,0x3456,0x4567]
But the following array would be returned by the function:
[0x1234,0,0,0]
This allows a malicious attester to bypass the quorum in the following way:

1. Generate N addresses that are not attesters where N is **quorum - 1**
2. Sign a valid batch using their own address
3. Add their own signature N times in the batch
4. Sign the batch using all other addresses that they generated
5. Add the other sigs to the batch
6. The returned array will have the following look: [(0xmalicious_attester),0,0,0,0,0,0] because they added first their own signed signature a couple of times which will create 0s in the array then the 'invalid' attesters that they generated would be trimmed out from the returned array
7. Generate a merkle tree multi proof using the returned attestersAsLeaves
8. Submit the batch and bypass the quorum

The last point works because it's possible to proof that the merkle tree contains value 0, as the default value of a leaf by definition is 0 and as such the attester can just verify their own address and a bunch of 0 values as leaves thus bypassing the quorum. The bypass can allow the malicious attester to also add other attesters as they can suggest the next committee and then approve them by themselves.

Remediation:

Instead of skipping duplicates and then trimming the array, simply append the attester to the array without any gaps then trim the array to the correct size thus not leaving any zeroes in the array.

T3RN1-16 | confirmBatchOrdersV3 doesn't work with ERC-20 tokens

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	High

Path:

contracts/avpBatchSubmitter.sol#L196-L206

Description:

In the **avpBatchSubmitter** contract, the **confirmBatchOrdersV3()** function always forwards **batch.totalAmount** of native tokens when settling the payout for each batch:

```
for (uint256 i = 0; i < batchCount; i++) {
    uint256 gasStartBatch = gasleft();
    BatchData memory batch = batchData[i];
->  ro.settlePayoutWithFeesCall{value: batch.totalAmount}(
        batch.totalAmount,
        batch.asset,
        batch.target,
        msg.sender,
        6,
        bytes32(0)
    );
```

This is incorrect behavior, as it should only forward native tokens when **batch.asset** is **address(0)**, which indicates that the order requires native tokens. Otherwise, the **remoteOrder** requires ERC-20 tokens from the sender. So this will cause the **confirmBatchOrdersV3** call to revert whenever the order uses ERC-20 tokens as the asset.

Remediation:

Only forward msg.value = **batch.totalAmount** when **batch.asset** is address(0).

T3RN1-3 | Incorrect implementation of EIP712 leads to signature replay attacks

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/attestationsVerifierProofs.sol

Description:

The **AttestationsVerifierProofs** batch verification mechanism violates **EIP-712** structured data signing standards by failing to implement proper domain separation. The message hash computation does not include the required domain separator containing **chainId** and **verifyingContract** parameters, directly contravening **EIP-712** specifications designed to prevent cross-contract and cross-chain signature replay attacks. This architectural flaw enables attackers to replay messages across different chains.

An attacker can extract a legitimately signed batch from the source chain and replay it on the target chain using identical calldata. Since the hashing domain remains consistent across deployments, signature verification succeeds despite the cross-chain context.

```
    function recoverSigner(bytes32 _messageHash, bytes memory signature) public
pure returns (address) {
        bytes32 r;
        bytes32 s;
        uint8 v;

        if (signature.length != 65) {
            return address(0);
        }

        assembly {
            r := mload(add(signature, 32))
            s := mload(add(signature, 64))
            v := byte(0, mload(add(signature, 96)))
        }

        if (v < 27) {
            v += 27;
```

```
        }

        if (v != 27 && v != 28) {
            return address(0);
        } else {
            bytes32 prefixedHash = keccak256(abi.encodePacked("\x19Ethereum
Signed Message:\n32", _messageHash));
            return ecrecover(prefixedHash, v, r, s);
        }
    }
```

Remediation:

To mitigate this vulnerability, implement a complete EIP-712 domain separator within the signed message hash. This ensures that batches signed for one contract or chain remain invalid when used on different contracts or chains.

T3RN1-15 | Batch Function for confirmOrderV3 Allow Double Execution by Different Executors

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/remoteOrder.sol#L328-L336

Description:

The **confirmOrderV3** function prevents double execution of orders by checking both:

executor confirmation ID:

```
bytes32 confirmationId = keccak256(abi.encode(id, target, amount, asset,
msg.sender));
```

Global order confirmation ID:

```
bytes32 orderConfirmationId = keccak256(abi.encode(id, target, amount, asset));
```

Both of these IDs are stored in the **orderPayloads** mapping:

```
orderPayloads[confirmationId] = id;
orderPayloads[orderConfirmationId] = id;
```

Which are used as validation for future calls to **confirmOrderV3**.

```
// First, check if the order is already confirmed
if (orderPayloads[confirmationId] != bytes32(0)) {
    revert("RO#7"); // RO#7: The operation was already confirmed
}

// Check if the order was delivered by another executor, preventing double
executions by 2 accounts
if (orderPayloads[orderConfirmationId] != bytes32(0)) {
    revert("RO#2"); // RO#2: The operation was already confirmed
}
```

This ensures the order is only confirmed once, by one executor.

Problem:
In the batched version, **confirmBatchOrdersV3**, only the executor-specific **confirmationId** is checked:

```
// Check if the order was delivered by another executor, preventing double
executions by 2 accounts
if (orderPayloads[orderConfirmationId] != bytes32(0)) {
    revert("RO#2"); // RO#2: The operation was already confirmed
}
```

Since it doesn't check the global **orderConfirmationId**, a different executor could still confirm the same order again, leading to possible double execution.

Remediation:

Consider setting the **orderConfirmationId** as well in the **confirmBatchOrdersV3.**

```
bytes32 confirmationId = keccak256(
    abi.encode(entries.ids[i], entries.targets[i], entries.amounts[i],
entries.assets[i], msg.sender)
);
++ bytes32 orderConfirmationId = keccak256(abi.encode(id, target, amount,
asset));

// Check if the order is already confirmed
if (ro.orderPayloads(confirmationId) != bytes32(0)) {
    continue;
}

++ if (ro.orderPayloads(orderConfirmationId) != bytes32(0)) {
++    continue;
++ }

ro.markOrderPayload(confirmationId, entries.ids[i]);
++ ro.markOrderPayload(orderConfirmationId, entries.ids[i]);
```

T3RN1-8 | distributeInflation Should Be Callable Weekly but Is Limited to Every Other Week

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/TRNInflation.sol#L99-L101

Description:

The **TRNInflation** contracts handle the distribution of inflation by calling the **distributeInflation** function once every week.

```
function distributeInflation() public {
    -- snip --

    uint256 weeksPassed = (block.timestamp - lastInflationTime) /
WEEKLY_INFLATION_INTERVAL;
    require(weeksPassed > 1, "Disallow repeated distributions");

    -- snip --

    lastInflationTime = block.timestamp;

.   -- snip --

    TRNInflationRewardsAddress.resetWeeklyCounters();
}
```

This function also resets the weekly counters in the **TRNInflationRewards** contract by calling **resetWeeklyCounters**.

```
function resetWeeklyCounters() public onlyTRNInflation {
    require(block.timestamp >= nextResetTime, "It is not time to reset yet");
    weeklyExecutorRewards = 0;
    weeklyUsersCounter = 0;
    weeklyAttestersCounter = 0;
    nextResetTime = block.timestamp + 1 weeks; // set the next reset time one
week from now emit WeeklyCountersReset(block.timestamp);
    emit WeeklyCountersReset(block.timestamp);
}
```

This function should be called every week but currently expects 2 weeks passed because of the **>** operator instead of **>=**

The issues arrises due the following check:

```
uint256 weeksPassed = (block.timestamp - lastInflationTime) /
WEEKLY_INFLATION_INTERVAL;
require(weeksPassed > 1, "Disallow repeated distributions");
```

Which only allows an update every other week instead of every week.

Remediation:

```
function distributeInflation() public {
...
    uint256 weeksPassed = (block.timestamp - lastInflationTime) /
WEEKLY_INFLATION_INTERVAL;
    -- require(weeksPassed > 1, "Disallow repeated distributions");
    ++ require(weeksPassed >= 1, "Disallow repeated distributions");
```

T3RN1-13 | leftThisWeek is reduced even when transfer fails or not reduced when finalReward equals leftThisWeek

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/bonusesL3.sol#L176-L199

Description:

In the **BonusesL3** contract, the function **applyBonusFromBid** is responsible for distributing rewards(bonuses), which are capped by the weekly stored amount in leftThisWeek.

Each distribution deducts the **finalReward** amount from **leftThisWeek** using the following logic:

```
if (finalReward < leftThisWeek) {
    leftThisWeek -= finalReward;
}
```

Problem 1: Off-by-One edge case

This check fails to account for the case where **finalReward == leftThisWeek**, which should be valid.

Example:

- If **leftThisWeek = 1 ether**
- And **finalReward = 1 ether**
- The condition **finalReward < leftThisWeek** is false, so **leftThisWeek** is not updated, even though the distribution may still succeed.

Problem 2: Deduction leftThisWeek Before Checking Distribution Success

The contract deducts **leftThisWeek** before confirming that the reward was actually distributed:

```
if (finalReward < leftThisWeek) {
    leftThisWeek -= finalReward;
}
_updateAssetAverage(assetId, amount);
bool distributed = distribute(beneficiary, finalReward);
```

The function doesn't revert because it doesn't require the transfer to succeed, it returns a boolean as status. If the transfer fails, no funds are actually sent, but **leftThisWeek** still gets reduced.

Problem 3: Still Calculates Rewards When **leftThisWeek == 0**

When **leftThisWeek** is zero, the contract still proceeds to calculate the base reward and bonus, even though no rewards can be distributed.

Remediation:

1. Use **>** to ensure the correct weekly cap check
2. Deduct **leftThisWeek** only after successful distribution
3. Don't calculate rewards if **leftThisWeek = 0**

```
    function applyBonusFromBid(
        uint32 assetId,
        uint256 amount,
        address beneficiary
    ) external onlyAuthorizedContract returns (bool) {
        checkResetCurrentPeriod();

++      _updateAssetAverage(assetId, amount);

++      if (leftThisWeek == 0) {
++         return false;
++      }

        uint256 reward = readCurrentBaseReward();

        uint256 bonus = _applyBonus(assetId, reward, amount);

        uint256 finalReward = reward + bonus;


--      if (finalReward < leftThisWeek) {
--          leftThisWeek -= finalReward;
--      }

++      if (finalReward > leftThisWeek) {
++         finalReward = leftThisWeek;
++      }
```

```
--      _updateAssetAverage(assetId, amount);

        // Distribute the reward
        bool distributed = distribute(beneficiary, finalReward);
        if (!distributed) {
            return false;
        }

++      leftThisWeek -= finalReward;


        emit BonusApplied(beneficiary, assetId, reward, bonus, finalReward);

        return distributed;
    }
```

T3RN1-22 | Using checkClaimPayoutBatch could result in users losing their claimable funds

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	High

Path:

contracts/claimerGMPV2.sol#L298

Description:

In the **claimerGMPV2** contract, the **checkClaimPayoutBatch()** function is used to check the claimable status of payouts and return information about the rewards that can be claimed from those payouts.

However, for each payout, this function always nullifies the **escrowOrdersPayloadHash** of its order ID by marking it as CLAIMED through a call to the **oneifyPayloadHash()** function.

```
function checkClaimPayoutBatch(
  RemoteOrder.Payout[] memory payouts,
  bytes32 _batchPayloadHash,
  bytes memory _batchPayload
) public returns (address[] memory, uint256[] memory, uint256) {
  address[] memory rewardAssets = new address[](payouts.length);
  uint256[] memory rewardAmounts = new uint256[](payouts.length);
  uint256 rewardAssetCount = 0;
  for (uint256 i = 0; i < payouts.length; ++i) {
      require(
          checkIsClaimable(
              payouts[i].id,
              payouts[i].rewardAsset,
              payouts[i].maxReward,
              payouts[i].settledAmount,
              msg.sender,
              payouts[i].orderTimestamp,
              _batchPayloadHash,
              _batchPayload
          ),
          "RO#15"
      );
```

```
        require(escrowGMP.getRemotePaymentPayloadHash(payouts[i].id) !=
bytes32(0), "RO#15");
        escrowGMP.oneifyPayloadHash(payouts[i].id);
        ...
```

This causes **escrowGMP** to mark the order as claimed, and the order will never be claimable again, since its **getRemotePaymentPayloadHash()** will return **1** (**CLAIMED_PAYLOAD**).

```
// Nullify Payload with One - One to be assumed it's claimed by executor
function oneifyPayloadHash(bytes32 orderId) external onlyOneOfOrderers {
    if (escrowOrderer == msg.sender) {
        escrowOrdersPayloadHash[orderId] = CLAIMED_PAYLOAD;
    } else {
        remotePaymentsPayloadHash[orderId] = CLAIMED_PAYLOAD;
    }
}
```

Remediation:

The **checkClaimPayoutBatch()** function should be a view function that does not modify storage, which can be achieved by removing the following line of code:

```
escrowGMP.oneifyPayloadHash(payouts[i].id);
```

T3RN1-17 | Similar entries in confirmBatchOrdersV3 should be merged into a single confirmationId

Acknowledged

Severity:	Probability:	Impact:
Low	Unlikely	Medium

Path:

contracts/avpBatchSubmitter.sol#L157-L178

Description:

In the **confirmBatchOrdersV3()** function, for each entry, it attempts to find an existing batch with the same target and asset to merge the entry into. This allows the user to use multiple entries for the same order.

```
// Try to find an existing batch entry with the same target and asset
bool found = false;
for (uint256 j = 0; j < batchCount; j++) {
    if (batchData[j].target == entries.targets[i] && batchData[j].asset ==
entries.assets[i]) {
        // Accumulate the amount for the existing target and asset pair
        batchData[j].totalAmount += entries.amounts[i];
        batchData[j].count++;
        found = true;
        break;
    }
}

// If no entry exists, create a new one
if (!found) {
    batchData[batchCount] = BatchData({
        target: entries.targets[i],
        asset: entries.assets[i],
        totalAmount: entries.amounts[i],
        count: 1
    });
    batchCount++;
}
```

After that, each batch contains the total amount from multiple similar entries and is used to settle the payout.

However, it uses the **confirmationId** of each entry before merging into a batch to mark the order payloads.

```
bytes32 confirmationId = keccak256(
    abi.encode(entries.ids[i], entries.targets[i], entries.amounts[i],
entries.assets[i], msg.sender)
);

// Check if the order is already confirmed
if (ro.orderPayloads(confirmationId) != bytes32(0)) {
    continue;
}

ro.markOrderPayload(confirmationId, entries.ids[i]);
```

It may mark multiple payloads with different **confirmationIds** for the same user and the same order. This can lead to confusion during the validation and commitment of payloads for users. Moreover, if a user submits two or more entries with the same amount, **ro.markOrderPayload()** call will revert due to the duplication of **confirmationId**, even though the batch intends to merge the entry amounts into the total amount of the order.

Remediation:

It should calculate the **confirmationId** and mark its payload for each batch after merging the similar entries.

T3RN1-5 | Certain chains will be impossible to whitelist due to type size limitations

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/remoteOrder.sol#L260
contracts/remoteOrder.sol#L39

Description:

In the **remoteOrder** contract when creating an order the contract checks whether the destination chain is supported by the protocol:

```
require(ensureDestAssetIsSupported(asset, destination), "RO#1");
```

To check whether the destination chain is supported or no the contract checks whether the network chain ID is inside of the **supportedNetworks** mapping or no. The mapping has the following definition:

```
mapping(bytes4 => bool) public supportedNetworks;
```

As such the max size of a supported chain ID is **bytes4** which will exclude several chains because a chain with the following chain ID already exists:
chains/_data/chains/eip155-868455272153094.json at master · ethereum-lists/chains

```
{
 "name": "Godwoken Testnet (V1)",
 "chain": "GWT",
 "rpc": ["https://godwoken-testnet-web3-v1-rpc.ckbapp.dev"],
 "faucets": ["https://homura.github.io/light-godwoken"],
 "nativeCurrency": {
   "name": "CKB",
   "symbol": "CKB",
   "decimals": 8
 },
 "infoURL": "https://www.nervos.org",
 "shortName": "gw-testnet-v1-deprecated",
```

```
  "chainId": 868455272153094,
  "networkId": 868455272153094,
  "slip44": 1,
  "status": "deprecated",
  "explorers": [
    {
      "name": "GWScan Block Explorer",
      "url": "https://v1.aggron.gwscan.com",
      "standard": "none"
    }
  ]
}
```

And many others can exist which may become a wanted network to be supported by the protocol but the protocol wouldn't be able to support the network because the chain ID will surpass the max type size of bytes4.

Remediation:

Change the variable type of the key of the mapping from **bytes4** to **uint256**.

T3RN1-6 | BRN to TRN migrator could result in lost funds due to silent fail

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

BRN2TRN.sol:brn2trn#L86-L99

Description:

The **brn2trn** function checks the sender's BRN balance and allows for some ratio to be converted to TRN, the native asset.

However, the function will always call **transferFrom** on BRN and transfer the user's assets to the burn address, while the **distribute** function could silently fail due to insufficient funds or a revert of the receive callback. In that case, it ignores the **false** return value and the user's funds would be lost.

```
    function brn2trn() public {
        // Check if inflation is on
        require(isOn, "Inflation is off");
        uint256 balance = BRNToken.balanceOf(msg.sender);
        uint256 ratio = BRN2TRNRatio;
        if (whitelistedAddresses[msg.sender]) {
            ratio = BRN2TRNWhitelistedRatio;
        }
        uint256 trnAmount = balance / ratio;
        // Burn BRN
        BRNToken.transferFrom(msg.sender, burntAddress, balance);
        // Distribute TRN via
        distribute(msg.sender, trnAmount);
    }

    function distribute(address to, uint256 amount) internal nonReentrant
returns (bool) {
        if (to == address(0)) {
            emit DistributionAttempt(to, amount, false, "Invalid address: zero
address");
           return false;
        }
        if (amount == 0) {
```

```
            emit DistributionAttempt(to, amount, false, "Invalid amount: must
be greater than zero");
            return false;
        }
        if (address(this).balance < amount) {
            emit DistributionAttempt(to, amount, false, "Insufficient
balance");
            return false;
        }
        (bool success, ) = to.call{value: amount}("");
        if (success) {
            emit DistributionAttempt(to, amount, true, "Success");
        } else {
            emit DistributionAttempt(to, amount, false, _getRevertMsg());
        }
        return success;
    }
```

Remediation:

We would recommend to perform a require check on the return value of **distribute** to ensure a revert in case the transfer failed:

```
require(distribute(msg.sender, trnAmount), "TRN transfer failed.");
```

T3RN1-9 | Lack of Setter Function for Whitelist Mapping

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

BRN2TRN.sol:brn2trn#L86-L99

Description:

BRN2TRN allows certain addresses to be whitelisted to receive a different **BRN2TRNRatio**.

```
function brn2trn() public {
      .. snip ..
    uint256 ratio = BRN2TRNRatio;
    if (whitelistedAddresses[msg.sender]) {
        ratio = BRN2TRNWhitelistedRatio;
    }
```

However there is currently no way to add an address to the **whitelistedAddresses** mapping.

Remediation:

Consider adding the following function.

```
function setWhitelistedAddress(address addr, bool isWhitelisted) external
onlyOwner {
    whitelistedAddresses[addr] = isWhitelisted;
}
```

T3RN1-12 | Payable function should reject unexpected ETH

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/remoteOrder.sol#L299-L318

Description:

In the **RemoteOrder** contract, users can settle payment in either native or ERC-20 tokens.

Because the function **confirmOrderV3()** is payable, it should have a check for msg.value must be 0 when its not a native transfer. Otherwise a user could mistakenly send native tokens along with a token transfer.

Remediation:

Consider adding the following check for payable functions that take either tokens or native tokens:

```
function confirmOrderV3(
    bytes32 id,
    address payable target,
    uint256 amount,
    address asset,
    uint32 nonce,
    address sourceAccount,
    bytes4 source
) public payable isOn returns (bool) {
.   -- snip --

    if (asset == address(0)) {
        require(msg.value == amount, "RO#2");
++  } else {
++      require(msg.value == 0, "RO#2");
++  }
```

T3RN1-18 | Misuse of Cloning in VestingFactory

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

contracts/vestingFactory.sol#L64-L65

Description:

The **VestingFactory** create vestingWallets, however each time it creates a new wallet, it deploys a new **CustomVestingWallet** contract just to use it as a template for cloning. This defeats the purpose of using **Clones.clone()**, which is supposed to save gas by reusing a single deployed implementation.

```
  address clone = Clones.clone(address(new CustomVestingWallet()));
  CustomVestingWallet(payable(clone)).initialize(beneficiary, startDate,
VESTING_DURATION_18_MONTHS);
```

Remediation:

The proper way to use **Clones.clone()** is:

1. Deploy one implementation contract of **CustomVestingWallet** (only once)
2. Store its address in a state variable (e.g., **vestingWalletImplementation**)
3. Use that single implementation to clone new wallets

T3RN1-19 | Price Calculation Ignores Token Decimals in OpenMarketPricer

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/openMarketPricer.sol

Description:

OpenMarketPricer.updatePrice computes price without adjusting for the decimals of **assetA** and **assetB**. If the caller of **storeQuote** supplies raw token amounts with differing decimal scales, the stored price will be skewed by the decimal mismatch.

```
    function updatePrice(bytes32 priceId, uint256 amountA, uint256 amountB,
uint256 currentTime) internal {
        PriceData storage priceData = perPairPrices[priceId];

        if (priceData.lastUpdateTime == 0) {
            // Initial price
            priceData.price = (amountA * 1e18) / amountB; // store price with
18 decimals
            priceData.volume = amountA;
        } else {
            uint256 timeDelta = currentTime - priceData.lastUpdateTime;

            if (timeDelta >= tickInterval) {
                // Calculate EWMA
                uint256 newPrice = (amountA * 1e18) / amountB;
                priceData.price = ((priceData.price * (1e18 - alpha)) +
(newPrice * alpha)) / 1e18;
                priceData.volume = ((priceData.volume * (1e18 - alpha)) +
(amountA * alpha)) / 1e18;
            } else {
                // Simply update the volume within the same tick interval
                priceData.volume += amountA;
           }
        }

        priceData.lastUpdateTime = currentTime;
    }
```

Remediation:

Add decimal normalisation inside the contract (e.g., keep a decimals mapping and convert amounts to a common 18-dec scale) or explicitly require callers to pass already-normalised amounts and enforce this with input validation.

T3RN1-21 | Inverted Time-Factor Causes Rewards to Decrease Near Week End

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/bonusesL3.sol

Description:

In **readCurrentBaseReward** (contracts/bonusesL3.sol), the variable **weekElapsed** is computed as the time remaining until the week ends.

```
    function readCurrentBaseReward() public view returns (uint256) {
        ...
        // Adjust reward scale based on how much time has passed
        uint256 timeFactor = (weekElapsed * 100) / 1 weeks; // Percentage of
the week elapsed (0-100%)
        if (timeFactor == 0) {
            timeFactor = 1;
        }
        if (timeFactor > 100) {
            timeFactor = 100;
        }

        // Dynamic slowdown factor based on how close we are to reaching the
weekly target
        uint256 slowdownFactor = (txCount * 100) / currentWeeklyTarget;

        // Ensure it never exceeds 100% scaling to prevent division issues
        if (slowdownFactor > 100) {
            slowdownFactor = 100;
        }

        // Adjust the base reward based on transaction count and supply
remaining
        uint256 baseReward = ((currentWeeklySupply / readWeeklyTarget()) * (100
- slowdownFactor) * timeFactor) /
            (100 * 100);
        ...
    }
```

The subsequent comment and logic treat this value as “percentage of the week elapsed,” resulting in a smaller **timeFactor** as the week progresses. Consequently, the base reward drops when the week is close to expiring, which is the opposite of the intended incentive model (higher rewards near deadline).

Remediation:

Calculate the actual elapsed time or invert the factor.

```
uint256 weekElapsed = block.timestamp - currentWeekStartDate;
uint256 timeFactor  = (weekElapsed * 100) / 1 weeks;
```

T3RN1-23 | Order can be confirmed multiple times

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/remoteOrder.sol#L325-L326

Description:

In the **confirmOrderV3()** function, it attempts to prevent any subsequent confirmations of an order by using the **orderPayloads** storage:

```
bytes32 confirmationId = keccak256(abi.encode(id, target, amount, asset,
msg.sender));
bytes32 orderConfirmationId = keccak256(abi.encode(id, target, amount, asset));

// First, check if the order is already confirmed
if (orderPayloads[confirmationId] != bytes32(0)) {
    revert("RO#7"); // RO#7: The operation was already confirmed
}

// Check if the order was delivered by another executor, preventing double
executions by 2 accounts
if (orderPayloads[orderConfirmationId] != bytes32(0)) {
    revert("RO#2"); // RO#2: The operation was already confirmed
}

// Store the confirmationId before proceeding
orderPayloads[confirmationId] = id;
orderPayloads[orderConfirmationId] = id;
```

The **amount** value doesn't need to be fixed in the case of partial fulfillment (open market). The **confirmationId** also includes the amount in its hash, which allows different amounts to be confirmed for a single order. However, only one **amount** value can be committed to the payments payload hash of the **escrowGMP** contract via the **commitRemoteBeneficiaryPayloadOpenMarket()** function.

Therefore, only one amount value can be used to claim funds, and the other confirmations for that order will be lost.

```
function commitRemoteBeneficiaryPayloadOpenMarket(
    bytes32 orderId,
    address beneficiary,
    uint256 rewardSettled
) external onlyAttesters returns (bool) {
    // Update the payment payload (hash of the payload)
    bytes32 currentHash = remotePaymentsPayloadHash[orderId];
    if (currentHash == EMPTY_PAYLOAD) {
        emit GMPExpectedPayloadNotMatched(orderId, beneficiary, currentHash);
        return (false);
    }
    if (currentHash == bytes32(uint256(2))) {
        emit PresumablyLateExecution(orderId, beneficiary);
        return (false);
    }
    bytes32 newHash = keccak256(abi.encode(currentHash, beneficiary,
rewardSettled));
    remotePaymentsPayloadHash[orderId] = newHash;
    return (true);
}
```

Remediation:

The **amount** value should be removed from **confirmationId** and **orderConfirmationId** in the **confirmOrderV3()** function.

T3RN1-1 | Unused events

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

contracts/attestationsVerifierProofs.sol
contracts/avpBatchSubmitter.sol
contracts/remoteOrder.sol

Description:

Several events declared in **attestationsVerifierProofs**, **avpBatchSubmitter**, **remoteOrder** contracts are never used.

```
    event SignerEmitted(address indexed signer);
```

```
    event TestEvent(bool, bool, bytes32[] leaves, address[] addressesRecovered,
bytes32);
```

```
    event CommitmentApplied(
        bytes32 indexed batchHash,
        address indexed executor,
        bytes32 indexed attestingCommitteeHash
    );
```

```
    event CallCommitApplied(bytes32 indexed sfxId);
```

```
    event CallRevertApplied(bytes32 indexed sfxId);
```

```
    event SignerNotInCommittee(address indexed signer);
```

Remediation:

Remove unused event declarations or emit these events in the appropriate functions.

T3RN1-2 | BRN contract constructor ignores owner parameter

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

contracts/BRN.sol#L11-L13

Description:

The constructor of the BRN contract takes an **_owner** as parameter, but it ignores this and sets the owner to **msg.sender**.

```
    constructor(address _owner, string memory name, string memory symbol)
ERC20(name, symbol) {
        owner = msg.sender;
    }
```

Remediation:

If the parameter is to be ignored, it should have its name removed or consider implementing OpenZeppelin's **Ownable** pattern for standardized ownership management.

T3RN1-10 | Missing Zero-Check in setRatio May Lead to Division by Zero

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

In the logic where we calculate **trnAmount** by dividing a balance by a ratio:

```
uint256 trnAmount = balance / ratio;
```

there is a potential risk of a division by zero error if **ratio** is set to 0. To prevent this, we should ensure that the ratio is always greater than zero before it is set.

Currently, the **setRatio** function does not include this validation:

```
function setRatio(uint256 _ratio) public onlyOwner {
    BRN2TRNRatio = _ratio;
}
```

The **setWhitelistedRatio** function correctly includes a check:

```
function setWhitelistedRatio(uint256 _ratio) external onlyOwner {
    require(_ratio > 0, "ratio = 0");
    BRN2TRNWhitelistedRatio = _ratio;
}
```

Remediation:

Add a similar **require(_ratio > 0)** check in **setRatio** to avoid setting an invalid ratio.

T3RN1-11 | Duplicate call value amount check

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/remoteOrder.sol#L299-L351

Description:

In **RemoteOrder::comfirmOrderV3** there is a duplicate check for **msg.value == amount**.

```
function confirmOrderV3(
        bytes32 id,
        address payable target,
        uint256 amount,
        address asset,
        uint32 nonce,
        address sourceAccount,
        bytes4 source
    ) public payable isOn returns (bool) {
     -- SNIP --

        if (asset == address(0)) {
            require(msg.value == amount, "RO#2");
        }
      -- SNIP --

        if (asset == address(0)) {
            require(msg.value == amount, "RO#2");
        }

      -- SNIP --
    }
```

Remediation:

Remove one check for **msg.value == amount**.

T3RN1-20 | Inconsistent Action-Type IDs Between AttestationsVerifierProofs and ClaimerGMPV2

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/attestationsVerifierProofs.sol
contracts/claimerGMPV2.sol

Description:

AttestationsVerifierProofs encodes actions as: 2 = **EscrowCommitApplied**, 6 = **TransferCommitOpenMarket**.

ClaimerGMPV2's **verifyGMPPayloadInclusion** expects: 0 = **escrow commit**, 5 = **open-market commit**.

The mismatch is currently benign because these specific IDs are not used in the live claim/refund paths, but it could cause rejections or mis-interpretation if future logic relies on shared payloads.

```
    enum OperationType {
        TransferCommit,
        TransferRevert,
        EscrowCommitApplied,
        Mint,
        CallCommit,
        CircuitHeaderEncoded,
        TransferCommitOpenMarket
    }

    function decodeAndProcessPayload(bytes calldata payload) private returns
(bool) {
        require(payload.length > 0, "EMPTY_PAYLOAD");

        uint256 offset = 0;
        while (offset < payload.length) {
            uint8 opType = uint8(payload[offset]);
            offset += 1; // To move past the operation type byte
            if (opType == uint8(OperationType.TransferCommit)) {
            ...
```

```
            } else if (opType == uint8(OperationType.TransferRevert)) {
            ...
            } else if (opType == uint8(OperationType.EscrowCommitApplied)) {
            ...
            }
        }
        return true;
    }
```

```
    function verifyGMPPayloadInclusion(
        bytes calldata payload,
        bytes32 _id,
        address _beneficiary,
        uint256 _settledAmount,
        uint8 _actionType
    ) public pure returns (bool) {
        uint256 offset = 0;

        while (offset < payload.length) {
            uint8 actionType = uint8(payload[offset]);
            offset += 1;
            // If escrow commit action
            if (actionType == 0) {
            ...
            }
            // If escrow revert action
            else if (actionType == 1) {
            ...
            }
            // If Open-Market commit action
            else if (actionType == 5) {
            ...
            }
        }

        return false;
    }
```

Remediation:

Adopt a single, shared definition for action-type codes and update both contracts to reference it.


hexens
x
t3rn