Quantstamp

Powerloom L2

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	L2
Timeline	2025-01-19 through 2025-01-21
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Docs
Source Code	• https://github.com/PowerLoom/protocol-contracts #d6b1b65
Auditors	• Mostafa Yassin Auditing Engineer • Mustafa Hasan Senior Auditing Engineer • Tim Sigl Auditing Engineer • Paul Clemson Auditing Engineer

Documentation quality	Medium
Test quality	Medium
Total Findings	25 Fixed: 20 Acknowledged: 4 Mitigated: 1
High severity findings	2 Fixed: 2
Medium severity findings	6 Fixed: 6
Low severity findings	9 Fixed: 7 Acknowledged: 1 Mitigated: 1
Undetermined severity findings	3 Fixed: 1 Acknowledged: 2
Informational findings	5 Fixed: 4 Acknowledged: 1

Summary of Findings

Fix Review Update

The client fixed/mitigated all the high and medium severity issues. The fixes were collectively provided. in the commit `f0e7c3ac8fb2080b2a9e50f04c4dc15332e41049`, which might have included changes that were not audit related. Only changes related to the issues were reviewed.

Initial Audit

Powerloom is a protocol deployed on Prost chain, that aims to provide a reliable and up-to-date market of data for third-party applications and users. The protocol allows users to mint nodes that have the ability to take snapshots for a given set of projects. Then, a centralized sequencer will submit these snapshots on-chain. Validators then start submitting attestations to the different batches. The incentive for nodes is to receive available rewards once they reach a defined threshold of submissions. Nodes are modeled as ERC1155 tokens and can be burned and the owners can claim back their original investment.

The protocol relies on a sophisticated off-chain event-handling system, that is out of the scope of this audit, as well as its interactions with the contracts.

The audit identified 2 main issues of high severity as well as multiple medium and low severity issues. The first issue is related to allowing for multiple votes per validator, and the second is an incorrect accounting for node rewards. it is recommended to fix the issues before deployment.

ID	DESCRIPTION	SEVERITY	STATUS
POW-1	The Legacy Node Vesting Formula Ignores `legacyNodeTokensSentOnL1` Leading to Excessive Reward Distribution	● High ⓘ	Fixed
POW-2	Validators Can Submit Multiple Attestations to Maliciously Force Wrong Consensus	● High ⓘ	Fixed
POW-3	Unnecessary Nested Loop Causes Identical Logic to Occur `n` Times Leading to Repeat Event Emissions and a Potential DoS	● Medium ⓘ	Fixed
POW-4	Data Market Creation Limited to 255 Due to Incorrect Variable Type	● Medium ⓘ	Fixed
POW-5	Usage of `tx.origin` for Access Control Is Unsafe Due to Phishing Risk	● Medium ⓘ	Fixed
POW-6	`dayCounter` Not Updated in `forceSkipEpoch()` May Lead to DoS	● Medium ⓘ	Fixed
POW-7	Lack of Input Validation	● Medium ⓘ	Fixed
POW-8	Inaccurate Event Emission Can Desynchronize Off-Chain Components	● Medium ⓘ	Fixed
POW-9	Lack of Checks Against Cliff Period May Lead to DoS	● Low ⓘ	Fixed
POW-10	Inconsistent Rewards Distribution Due Complex `updateRewards()` Architecture	● Low ⓘ	Fixed
POW-11	Incorrect Block Time Assumptions Across Chains	● Low ⓘ	Acknowledged
POW-12	Missing Ordering Validation Between Submission Windows	● Low ⓘ	Fixed
POW-13	Use of `transfer()` Is Deprecated and Could Lead to Denial of Service	● Low ⓘ	Fixed
POW-14	Ownership Can Be Renounced	● Low ⓘ	Mitigated
POW-15	Data Market Owner Can Override Protocol State Address	● Low ⓘ	Fixed
POW-16	Improper Snapshotter Address Handling when Disabling Nodes	● Low ⓘ	Fixed
POW-17	`PowerloomDataMarket.maxSnapshotsCid()` Function Returns Unfinalized CID Despite Implying Consensus	● Low ⓘ	Fixed
POW-18	Missing Incentives After Daily Quota Creates Operational Risk	● Informational ⓘ	Acknowledged
POW-19	Potential Incorrect Removal From `allSnapshotters` Mapping in `SnapshotterState._disableNode()`	● Informational ⓘ	Fixed

ID	DESCRIPTION	SEVERITY	STATUS
POW-20	Initialization Logic Should Use Constructor Instead of `initialize()` Function	● Informational ⓘ	Fixed
POW-21	Redundant Event Emissions in Role Updates	● Informational ⓘ	Fixed
POW-22	Day Counter Can Become Desynchronized when Epochs per Day Is Updated	● Informational ⓘ	Fixed
POW-23	Values Not Reset in Case a Finalization Is Not Reached	● Undetermined ⓘ	Acknowledged
POW-24	Uncleared State Variable on Burning	● Undetermined ⓘ	Acknowledged
POW-25	Unused state variable	● Undetermined ⓘ	Fixed

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer

Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
   1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

Repo: https://github.com/PowerLoom/protocol-contracts(d6b1b65286590b9b226e796750222c34e49a4187) Files:

- hardhat/contracts/DataMarket.sol
- hardhat/contracts/DataMarketFactory.sol
- hardhat/contracts/ProtocolState.sol
- hardhat/contracts/SnapshotterState.sol

Files Excluded

All other files, as well as the off-chain components and their interactions with the contracts.

Operational Considerations

1. The sequencer must be inherently trusted within the system:
    - They must honestly build snapshot batches.
    - They must verify the content of each snapshot is correct.
    - They must correctly distribute snapshotter rewards.
2. The owner must ensure that the `ProtocolState` and `SnapshotterState` contracts hold enough tokens to fulfill user rewards across the two contracts.
3. Snapshooters are rewarded for submitting snapshots only until they reach the `dailySnapshotQuota`. After this, they will no longer be incentivized to create snapshots until the following day. This could cause issues where snapshooters all reach the quota early in the day and then stop processing snapshots until the next day begins.

Key Actors And Their Capabilities

Protocol Admin

The protocol admin can set and update major states in the contract through the following functions:

- DataMarket.initialize()
- DataMarket.updateProtocolState()
- DataMarket.updateEpochManager()
- ProtocolState.updateDataMarketFactory()
- ProtocolState.updateSnapshotterState()
- DataMarket.updateDaySize()
- DataMarket.updateDailySnapshotQuota()
- DataMarket.updateRewardPoolSize()
- DataMarket.updateMinAttestationsForConsensus()
- DataMarket.updateBatchSubmissionWindow()
- DataMarket.updateSnapshotSubmissionWindow()
- DataMarket.updateAttestationSubmissionWindow()
- DataMarket.updateSnapshotSubmissionWindow()
- SnapshotterState.updateMaxSupply()
- SnapshotterState.setMintStartTime()
- SnapshotterState.setSnapshotterAddressChangeCooldown()
- SnapshotterState.updateNodePrice()
- SnapshotterState.setSnapshotterTokenClaimCooldown()

Sequencer

The sequencer is a centralized entity that can submit batches for attestations through the `submitSubmissionBatch()` function. It can also end batch submission through the `endBatchSubmissions()` function.

For reward logic, it can update the number of nodes per day through the `updateEligibleNodesForDay()` function, and it can call the `updateRewards()` to distribute rewards to eligible nodes.

Validators

Validators can call the `submitBatchAttestation()` function to cast votes for a specific `finalizedCidsRootHash`

Node Owners

Node owners can mint and burn nodes through the `_mintNode` and `_burnNode()` functions. Nodes are modeled as ERC1155 tokens and are responsible for taking regular snapshots.

Findings

POW-1

The Legacy Node Vesting Formula Ignores `legacyNodeTokensSentOnL1` Leading to Excessive Reward Distribution

● High ⓘ Fixed

File(s) affected: `SnapshotterState.sol`

Description: Owners of legacy nodes who have provided KYC details can claim node token rewards after burning their node. The intention of the protocol is to reward the owners `legacyNodeValue` subtracted by `legacyTokensSentOnL1`. These tokens are partly transferred immediately upon burning the tokens (the `initialClaim` amount) with the remaining being vested over time. This remaining amount is set as the `tokensAfterInitialClaim` field in `nodeIdToVestingInfo` here:

```
nodeIdToVestingInfo[_nodeId] = LegacyNodeVestingInfo(
    msg.sender,
    initialClaim,
->  (legacyNodeValue - legacyTokensSentOnL1 - initialClaim),
    0,
    block.timestamp
);
```

However, all calculations when determining a user's vested amounts do not consider the `legacyTokensSentOnL1` as highlighted by this line in the `vestedLegacyNodeTokens()` function:

```
uint256 totalTokens = legacyNodeValue - initialClaim;
```

Therefore users will be able to claim the entire `legacyNodeValue` once vesting is complete regardless of how many tokens have already been sent to them on the layer one chain, resulting in legacy node owners each receiving a larger amount of tokens than is intended by the protocol.

Recommendation: Consider adjusting the logic when calculating the amounts of tokens available during vesting to handle the `legacyTokensSentOnL1`.

POW-2

Validators Can Submit Multiple Attestations to Maliciously Force Wrong Consensus

● High ⓘ Fixed

File(s) affected: `DataMarket.sol`

Description: The function `submitBatchAttestation()` allows a validator to vote on a given batchCid while providing a `finalizedCidsRootHash`. However, it does not guard against a validator voting again on either a different or the same hash in the same epoch.

This means a single validator can vote multiple times to force a batch to reach the finalization stage.

Exploit Scenario:
This proof of concept shows that a validator can vote twice:

```
 it("Testing Attestation Batch flow", async function () {
            const batchCid = "QmbWqxBEKC3P8tqsKc98xmWNzrzDtRLMiMPL8wBuTGsMnX";
            const batchId = 1;
            const projectIds = ["test-project-1", "test-project-2"];
            const snapshotCids = ["QmbWqxBEKC3P8tqsKc98xmWNzrzDtRLMiMPL8wBuTGsMnR",
"QmbWqxBEKC3P8tqsKc98xmWNzrzDtRLMiMPL8wBuTGsMnS"];

            const finalizedRootHash = ethers.encodeBytes32String("test-hash");
            const epochId = currentEpoch.epochId;

            // set otherAccount1 as a sequencer
            const role = 1
            await proxyContract.updateAddresses(
                dataMarket1.target,
                role,
                [otherAccount1.address],
                [true],
            );
            await expect(proxyContract.updateBatchSubmissionWindow(dataMarket1.target,
10)).to.not.be.reverted;

            const blockTimestamp = await time.latest();
            await expect(proxyContract.connect(otherAccount1).submitSubmissionBatch(
                dataMarket1.target,
                batchCid,
                epochId,
                projectIds,
                snapshotCids,
                finalizedRootHash
            )).to.emit(proxyContract, "SnapshotBatchSubmitted")
              .withArgs(dataMarket1.target, batchCid, epochId, blockTimestamp + 1);

            const project1Status = await proxyContract.snapshotStatus(dataMarket1.target,
projectIds[0], epochId);
            const project2Status = await proxyContract.snapshotStatus(dataMarket1.target,
projectIds[1], epochId);

            expect(project1Status.status).to.equal(0);
            expect(project2Status.status).to.equal(0);
            expect(project1Status.snapshotCid).to.equal(snapshotCids[0]);
            expect(project2Status.snapshotCid).to.equal(snapshotCids[1]);

            const role2 = 0

            await proxyContract.updateAddresses(
                dataMarket1.target,
                role2,
                [otherAccount1.address, otherAccount2.address, otherAccount3],
                [true, true, true],
            );

            let otherHash = ethers.encodeBytes32String("test-hash1")
            let otherHash1 = ethers.encodeBytes32String("test-hash2")
            await proxyContract.connect(otherAccount1).submitBatchAttestation(dataMarket1.target,
batchCid, epochId, finalizedRootHash)
            await proxyContract.connect(otherAccount1).submitBatchAttestation(dataMarket1.target,
batchCid, epochId, finalizedRootHash)
        });
```

Recommendation: Make use of the `mapping attestationsReceived[batchCid][tx.origin]` to prevent multiple voting. This can look like this:

`require(!attestationsReceived[batchCid][tx.origin])`

POW-3

Unnecessary Nested Loop Causes Identical Logic to Occur `n` Times Leading to Repeat Event Emissions and a Potential DoS

● Medium ⓘ Fixed

File(s) affected: `DataMarket.sol`

Description: When finalizing a snapshot batch in the `DataMarket` contract, the `finalizeSnapshotBatch()` function loops over all the projects in `batchCidToProjects[batchId]` to update the necessary state for these projects and emits a `SnapshotFinalized` event. However, this loop is wrapped inside an identical loop of the same length, meaning the state updates and event emission for each project will be repeated `n` times. These incorrect event emissions could lead to incorrect actions being taken by the off-chain processes of the protocol.

Additionally, this `O(n²)` operation will mean that if the number of projects in the `batchCidToProjects[batchId]` grows, the function may become uncallable if the gas required becomes greater than the chain's block gas limit.

Recommendation: Consider removing the unnecessary outer loop.

POW-4
Data Market Creation Limited to 255 Due to Incorrect Variable Type

● Medium ⓘ Fixed

File(s) affected: `ProtocolState.sol`

Description: In `PowerloomProtocolState`, the `dataMarketCount` is declared as `uint8` limiting it to store `255` at maximum.

When creating a new data market through `createDataMarket()`, the counter is incremented. If 255 data markets are created, incrementing `dataMarketCount` will cause an overflow and revert due to Solidity 0.8.x's built-in overflow protection. This unnecessarily limits the maximum number of data markets that can exist in the protocol. New contracts need to be deployed when reaching this limit to still be able to create data markets.

Recommendation: Change `dataMarketCount` to `uint256` to allow for a significantly larger number of data markets to be created.

POW-5
Usage of `tx.origin` for Access Control Is Unsafe Due to Phishing Risk

● Medium ⓘ Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: The contract uses `tx.origin` for authorization across multiple access control modifiers:

```
modifier onlyOwnerOrigin() {
    require(owner() == tx.origin, "E03");
}

modifier onlySequencer() {
    require(sequencerSet.contains(tx.origin), "E04");
}

modifier onlyEpochManager() {
    require(epochManager == tx.origin, "E05");
}

modifier onlyOwnerOrAdmin() {
    require(owner() == tx.origin || adminSet.contains(tx.origin), "E08");
}
```

Using `tx.origin` for access control is dangerous as it makes privileged functions vulnerable to phishing attacks. A malicious contract could trick authorized users (owner, sequencers, epoch manager, admins) into interacting with it, which would then forward calls to the target contract. Since `tx.origin` would still reflect the authorized user's address, all access controls would be bypassed.

Recommendation: Replace all `tx.origin` checks with `msg.sender` and restructure the contract architecture to properly handle authorization through protocol state changes. Consider:

1. Use `msg.sender` instead of `tx.origin` throughout all modifiers
2. Either pass sender addresses explicitly through protocol state functions
3. Or authorize the `PowerloomProtocolState` to only access the certain `PowerloomDataMarket` function by a a modifier like `onlyProtocolState` and protect the `PowerloomProtocolState` by a `onlyOwner` modifier.

POW-6
`dayCounter` Not Updated in `forceSkipEpoch()` May Lead to DoS

● Medium ⓘ Fixed

File(s) affected: `DataMarket.sol`

Description: The function `releaseEpoch()` increments the `dayCounter` value if the `epochIdCounter` overflows to the next day. The same logic does not exist in `forceSkipEpoch()` which can be called by the owner to skip a number of epochs, which may lead to a DoS when the sequencer calls `updateRewards()`.

Recommendation: Update the day counter with the number of skipped days.

POW-7 Lack of Input Validation

● Medium ⓘ Fixed

File(s) affected: `DataMarket.sol`, `ProtocolState.sol`, `SnapshotterState.sol`

Description: The following functions operate on addresses without checking them against the zero address:

- DataMarket.initialize()
- DataMarket.updateIolState()
- DataMarket.updateEpochManager()
- ProtocolState.updateDataMarketFactory()
- ProtocolState.updateSnapshotterState()

Additionally, the following functions do not validate their inputs against minimum or maximum values, which may lead to DoS cases or otherwise unwanted or unexpected outcomes:

- DataMarket.updateDaySize()
- DataMarket.updateDailySnapshotQuota()
- DataMarket.updateRewardPoolSize()
- DataMarket.updateMinAttestationsForConsensus()
- DataMarket.updateBatchSubmissionWindow()
- DataMarket.updateSnapshotSubmissionWindow()
- DataMarket.updateAttestationSubmissionWindow()
- DataMarket.updateSnapshotSubmissionWindow()
- DataMarket.updateSnapshotSubmissionWindow()
- DataMarket.updateSnapshotSubmissionWindow()
- SnapshotterState.updateMaxSupply()
- SnapshotterState.setMintStartTime()
- SnapshotterState.setSnapshotterAddressChangeCooldown()
- SnapshotterState.updateNodePrice()
- SnapshotterState.setSnapshotterTokenClaimCooldown()

Recommendation: Perform proper validation on passed values before operating on them.

POW-8 Inaccurate Event Emission Can Desynchronize Off-Chain Components

● Medium ⓘ Fixed

File(s) affected: `ProtocolState.sol`, `DataMarket.sol`

Description: In the case more than half of all validators are offline and consensus cannot be finalized the function `PowerloomDataMarket.forceCompleteConsensusAttestations()` can be used by the owner to complete the consensus without the vote of the majority of the validators.

Still, other requirements such as having a root hash with more votes than on other root hashes are mandatory for the owner to force complete the consensus. Due to a bug in the function logic, the consensus might not be completed but it will still be recorded in the off-cain system as completed.

The bug is in the if-clause that wraps the function body:

```
function forceCompleteConsensusAttestations(
        string memory batchCid,
        uint256 epochId
    ) public onlyOwnerOrigin returns (
        bool TRIGGER_BATCH_RESUBMISSION
    ) {
        if (checkDynamicConsensusAttestations(batchCid, epochId)) {
            TRIGGER_BATCH_RESUBMISSION = finalizeSnapshotBatch(batchCid, epochId);
        }
    }
```

If `checkDynamicConsensusAttestations()` returns false (meaning consensus requirements were not met), the function `forceCompleteConsensusAttestations()` returns the default value `false` of its boolean return value `TRIGGER_BATCH_RESUBMISSION`. Then in this function:

```
  function forceCompleteConsensusAttestations(PowerloomDataMarket dataMarket, string memory
batchCid, uint256 epochId) public {
        bool TRIGGER_BATCH_RESUBMISSION = dataMarket.forceCompleteConsensusAttestations(batchCid,
epochId);
```

```
            if(TRIGGER_BATCH_RESUBMISSION){
                emit TriggerBatchResubmission(address(dataMarket), epochId, batchCid,
block.timestamp);
            } else {
                _finalizeSnapshotBatchEvents(dataMarket, batchCid, epochId);
                emit SnapshotBatchFinalized(address(dataMarket), epochId, batchCid, block.timestamp);
            }
        }
```

`TRIGGER_BATCH_RESUBMISSION` will be false, which means it will invoke the `_finalizeSnapshotBatchEvents()` function will emit events, signaling to the off-chain components that the batch was finalized, while it is not finalized on the chain.

Exploit Scenario:

1. Owner calls forceCompleteConsensusAttestations() on a batch that:
   - Has fewer attestations than minAttestationsForConsensus
   - Is outside the submission window
   - Has already been finalized
   - Has tied attestations between different root hashes
2. checkDynamicConsensusAttestations() returns false
3. Function returns false, signaling successful finalization
4. PowerloomProtocolState.forceCompleteConsensusAttestations() interpret this as consensus being reached and emitting corresponding events.

Recommendation: It is recommended to use enums in order to handle the different cases.

POW-9 Lack of Checks Against Cliff Period May Lead to DoS

• Low ⓘ Fixed

File(s) affected: `SnapshotterState.sol`

Description: The function `configureLegacyNodes()` does not validate that the value of `legacyNodeVestingDays` is greater than that of `legacyNodeCliff`. In case the values the opposite is true or the values are equal, the line calculating `tokensVested` in `vestedLegacyNodeTokens()` will revert, causing a DoS case.

Recommendation: Include a check for the values.

POW-10 Inconsistent Rewards Distribution Due Complex `updateRewards()` Architecture

• Low ⓘ Fixed

File(s) affected: `ProtocolState.sol`

Description: Snapshotters receive rewards in the protocol by the sequencer calling the `updateRewards()` function in the `ProtocolState` contract. In the event that rewards are updated for a data market more than once on a given `day` it becomes likely that the rewards distributed for that day will be larger than the data market's intended `rewardPoolSize`. This is the case because the `eligibleNodes` field passed initially will not be changed the second time this function. Therefore `rewards` can end up being split in larger than intended amounts, causing the `dataMarket` in question to assign users rewards that have been marked as rewards for other data markets.

```
uint256 rewards = dataMarket.rewardPoolSize() / dataMarket.eligibleNodesForDay(day);
```

While ultimately this is in control of the sequencer to handle correctly, the architecture of the system significantly increases the likelihood that errors occur, either causing users to receive larger than intended reward amounts or forcing the protocol to choose not to send users earned rewards to avoid this from happening.

Recommendation: Consider allowing the protocol to calculate how many nodes are eligible for rewards without requiring input from the sequencer. Additionally given how likely it is for an error to occur if the `updateRewards()` function is called twice in the same day for a given `dataMarket`, sufficient safeguards should be in place at the contract level to avoid this from happening.

POW-11 Incorrect Block Time Assumptions Across Chains

• Low ⓘ Acknowledged

File(s) affected: `PowerloomDataMarket.sol`

Description: The `PowerloomDataMarket` contract assumes a fixed block time when calculating epochs per day:

```
epochsInADay = DAY_SIZE / (SOURCE_CHAIN_BLOCK_TIME * epochSize);
```

The calculation assumes block times are fixed by using constant `SOURCE_CHAIN_BLOCK_TIME`, but block times vary significantly across different blockchain networks and consensus mechanisms:

- Ethereum on PoS has ~12 second target block times with empty slots possible
- Block times are not guaranteed to be constant and can fluctuate based on network conditions

This fixed assumption could lead to incorrect epoch timing calculations and rewards distribution when deployed on chains with different block time characteristics.

Recommendation: Consider implementing a more flexible mechanism that can adapt to different chain characteristics and block time variations rather than relying on fixed block time assumptions.

POW-12 Missing Ordering Validation Between Submission Windows

• Low ⓘ Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: `PowerloomDataMarket`'s window update functions lack validation to maintain the required sequencing:

```
updateBatchSubmissionWindow()
updateSnapshotSubmissionWindow()
updateAttestationSubmissionWindow()
```

The submission windows must follow a specific order:

- Snapshot window should be shorter than batch window
- Batch window should be shorter than attestation window
- Attestation window must be longest to allow validators time to submit

Current implementation allows windows to be set in any order, which could:

- Prevent validators from submitting attestations if attestation window is too short
- Create timing conflicts between different submission phases
- Break the intended sequence of the submission process

Recommendation: Add validation checks to ensure proper ordering between submission windows whenever any window duration is updated (snapshot < batch < attestation).

POW-13 Use of `transfer()` Is Deprecated and Could Lead to Denial of Service

• Low ⓘ Fixed

File(s) affected: `Multiple contracts`

Description: The contracts use `transfer()` to send the native token, which forwards a fixed amount of gas (2300) that could be insufficient.

This pattern appears in multiple places throughout the codebase and could cause transactions to fail if:

- Recipient is a contract with a gas-intensive receive/fallback function
- Gas costs change due to future hardforks

This could lead to rewards becoming stuck and users unable to claim them.

Recommendation: Replace `transfer()` with `call()` and add checks for return values to ensure successful execution.

POW-14 Ownership Can Be Renounced

• Low ⓘ Mitigated

Update
Now uses two steps ownership transfer, but ownership can still be renounced.

File(s) affected: `DataMarket.sol`, `SnapshotterState.sol`, `ProtocolState.sol`

Description: Currently, the ownership `DataMarket`, `SnapshotterState` and `ProtocolState` can all be renounced, and the contracts can be left with no owner.

Recommendation: It is recommended to use `Ownable2Step` or override the `renounceOwnership()` method to prevent renouncing.

POW-15
Data Market Owner Can Override Protocol State Address

• Low ⓘ Fixed

File(s) affected: `DataMarket.sol`

Description: The `updateProtocolState()` function allows a data market owner to set the `protocolState` address to that of a contract under their control, which may allow them to bypass existing checks such as the `isActive` modifier.

Recommendation: Remove the function.

POW-16
Improper Snapshotter Address Handling when Disabling Nodes

• Low ⓘ Fixed

File(s) affected: `SnapshotterState.sol`

Description: When a node is burned, a call to `_disableNode()` is made. The function removes the snapshotter address of the node from the `allSnapshotters` mapping without checking if `snapshotterToNodeIds[node.snapshotterAddress].length() == 0` as done in `_assignSnapshotterToNode()`. Additionally, the `snapshotterToNodeIds` is not updated to remove the node ID from the array against the snapshotter's address.

Recommendation: - Perform the check to make sure the snapshotter address should be removed.

- Remove the node ID from the array.

POW-17
`PowerloomDataMarket.maxSnapshotsCid()` Function Returns Unfinalized CID Despite Implying Consensus

• Low ⓘ Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: The `PowerloomDataMarket.maxSnapshotsCid()` function's name and documentation suggest returns are consensus-backed, but this isn't enforced:

```
/** @dev Retrieves the snapshot CID with the maximum consensus for a given project and epoch */
```

Despite implying "maximum consensus", the function:

- Returns CIDs immediately after sequencer submission before any validator attestations
- Returns potentially incorrect CIDs when consensus was not reached and batch was resubmitted
- Provides no indication to caller whether the returned CID is finalized or still pending
- Has misleading natspec that suggests strong consensus when none may exist

This could cause external systems to treat unverified CIDs as having achieved consensus when they have not.

Recommendation: Either rename the function and update documentation to accurately reflect that it returns the current CID regardless of consensus state, or add a return value indicating consensus status.

POW-18
Missing Incentives After Daily Quota Creates Operational Risk

• Informational ⓘ Acknowledged

File(s) affected: `PowerloomDataMarket.sol`

Description: The `PowerloomDataMarket.updateRewards()` function lacks incentives for snapshotters to continue submitting snapshots after reaching `dailySnapshotQuota` because everyone reaching the threshold is rewarded the same.

Exploit Scenario:
1. Snapshotter reaches the dailySnapshotQuota and receives their reward share
2. With no additional rewards available, snapshotter stops submitting snapshots
3. If enough snapshotters follow this behavior, the network may fail to reach consensus on remaining snapshots

Recommendation: Implement a gradual reward system that incentivizes continued snapshotting even after reaching the `dailySnapshotQuota`.

POW-19
Potential Incorrect Removal From `allSnapshotters` Mapping in `SnapshotterState._disableNode()`

● Informational ⓘ Fixed

File(s) affected: `SnapshotterState.sol`

Description: Within the `_disableNode()` function, when a node is burned the `snapshotterAddress` that is linked to it is removed from the `allSnapshotters` mapping in the following line:

```
allSnapshotters[node.snapshotterAddress] = false;
```

However, there is no guarantee that the node being burned is the only node linked to this particular snapshotter address. This differs from the logic within the `_assignSnapshotterToNode()` function, which checks that the snapshotter address has no other nodes tied to it before removing it from the `allSnapshotters` mapping.

Recommendation: Consider adding the same logic that is present in the `_assignSnapshotterToNode()` function to ensure a snapshotter has no other nodes linked to it before removing it from the `allSnapshotters` mapping:

```
if (snapshotterToNodeIds[node.snapshotterAddress].length() == 0) {
    allSnapshotters[node.snapshotterAddress] = false;
    emit allSnapshottersUpdated(node.snapshotterAddress, false);
}
```

POW-20
Initialization Logic Should Use Constructor Instead of `initialize()` Function

● Informational ⓘ Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: The `PowerloomDataMarket` contract uses an `initialize()` function instead of a constructor, but is not an upgradeable contract.

For non-upgradeable contracts, initialization values should be set in the constructor to prevent the contract from being used in an uninitialized state. While there is an initialization check, constructor initialization provides better guarantees.

Recommendation: Convert the initialize() function into a constructor since the contract is not upgradeable.

POW-21 Redundant Event Emissions in Role Updates

● Informational ⓘ Fixed

File(s) affected: `PowerloomDataMarket.sol`,

Description: The `PowerloomDataMarket.updateAddresses()` function emits role update events even when no state change occurs:

- `ValidatorsUpdated` event emitted when adding an existing validator
- `ValidatorsUpdated` event emitted when removing a non-existent validator
- `SequencersUpdated` event emitted when adding an existing sequencer
- `SequencersUpdated` event emitted when removing a non-existent sequencer
- `AdminsUpdated` event emitted when adding an existing admin
- `AdminsUpdated` event emitted when removing a non-existent admin

This could mislead off-chain systems tracking these events into thinking role changes occurred when the contract state remained unchanged.

Recommendation: Only emit role update events when the address set is actually modified by checking the return value of `add()` and `remove()` functions.

POW-22 Day Counter Can Become Desynchronized when Epochs per Day Is Updated

● Informational ⓘ Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: The `PowerloomDataMarket.releaseEpoch()`'s day tracking mechanism in can break when `epochsInADay` is decreased:

When `epochsInADay` is reduced (e.g., from 100 to 40), the modulo check may miss the start of new days:

- With epochIdCounter = 99 and epochsInADay = 100, next increment would trigger new day
- If epochsInADay is changed to 40, increment to 100 won't trigger new day since 100 % 40 ≠ 1
- This extends the day length despite shorter epochs per day configuration

This could lead to inaccurate day counting and affect reward distributions that depend on daily periods.

Recommendation: Implement a mechanism to handle day counter recalculation when `epochsInADay` is modified to maintain accurate day tracking.

POW-23 Values Not Reset in Case a Finalization Is Not Reached

● Undetermined ⓘ Acknowledged

File(s) affected: `DataMarket.sol`

Description: The function `submitSubmissionBatch()` sets the `projectFirstEpochId` and `lastSequencerFinalizedSnapshot` for the given project ID and epoch, however the values are not reset in case validators do not reach consensus and a finalization never happens for the given epoch.

Recommendation: Reset the values when a finalization is not reached.

POW-24 Uncleared State Variable on Burning

● Undetermined ⓘ Acknowledged

File(s) affected: `SnapshotterState.sol`

Description: The function `burnNode()` allows a token owner to burn their node and start the reward claiming flow. However, when a node is minted, the mapping `nodeIdToOwner` maps the `nodeId` to the owner address, but when the token is burned, this mapping is not reset.

Recommendation: Currently, this does not have an impact. But since the protocol is upgradeable, it is recommended to reset this mapping

POW-25 Unused state variable

● Undetermined ⓘ Fixed

File(s) affected: `DataMarket.sol`

Description: The function `endBatchSubmissions()` is called by the sequencer and is supposed to signal the ending of batch submissions for an epoch by modifying the `epochIdToBatchSubmissionsCompleted[epochId]` variable. However, this variable is not used anywhere.

Recommendation: Consider using this variable where it is appropriate.

Auditor Suggestions

S1 Gas Optimizations

Fixed

File(s) affected: `DataMarket.sol`, `SnapshotterState.sol`

Description:

- The `ROLE` value in `DataMarket.updateAddresses()` is redundant as it is the same as the `role` parameter passed into it. Consider removing it.
- In `SnapshotterState.claimableNodeTokens()` the require statements checking `nodeIdToOwner[_nodeId] == msg.sender` and `node.burnedOn > 0` can be moved to the top of the function call since they are performed in all branches.
- The require statements in both of the `else` blocks in `SnapshotterState.claimNodeTokens()` are redundant and should be removed as they are already performed in `claimableNodeTokens()`.
- Unless the emitted event is required, the `SnapshotterState.deposit()` function can be removed since the contract already has a `receive()` function.

Recommendation: Implement the optimizations.

S2 Use ERC-721 Instead of ERC-1155 to Simplify Contract

Acknowledged

Description: No features of the ERC-1155 contract are used. The simpler ERC-721 standard seems to fulfill all the requirements of the protocol. Consider switching over to the ERC-721 implementation of OpenZeppelin.

S3 Different Solidity Versions (0.8.24, ^0.8.20)

Fixed

File(s) affected: `PowerloomDataMarket.sol`, `PowerloomNodes.sol`

Description: Different Solidity versions are used in the contracts. Consider using a single Solidity compiler version for consistency

Recommendation: Consider using a single Solidity compiler version for consistency

S4 Unlocked Pragma

Fixed

Related Issue(s): SWC-103

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.8.*`. The caret (`^`) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, we recommend to remove the caret to lock the file onto a specific Solidity version.

S5 Cache Storage Variables in Memory Variables in for-Loops

Fixed

File(s) affected: `PowerloomProtocolState.sol`

Description: Some for-loops in the contracts read iterator variables from storage. Reading from storage in every loop iteratio is expensive. Saving the variable to memory before the loop minimizes the gas consumption.

A few examples are:

- `PowerloomProtocolState._finalizeSnapshotBatchEvents()`
  - `for (uint i = 0; i < dataMarket.batchCidDivergentValidatorsLen(batchCid); i++)`
  - `for (uint i = 0; i < dataMarket.batchCidToProjectsLen(batchCid); i++)`

S6 Improve Variable Naming in `EpochInfo` Struct

Acknowledged

File(s) affected: `PowerloomDataMarket.sol`

Description:

Consider renaming variables in `PowerloomDataMarket.EpochInfo` struct for better clarity:

blocknumber → startBlockNumber epochEnd → endBlockNumbe

S7 Remove Redundant Default Value Initialization

Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: Consider removing explicit initialization of variables to their default values since Solidity automatically initializes them:

- `PowerloomDataMarket.isInitialized = false`
- `PowerloomDataMarket.epochIdCounter = 0`

S8 Improve Readability of Consensus Check Logic

Fixed

File(s) affected: `PowerloomDataMarket.sol`

Description: The `PowerloomDataMarket.checkDynamicConsensusAttestations()` function contains complex validation logic in a single if statement. Consider breaking down conditions into descriptively named variables or functions to improve code readability:

```
bool isAttested = batchCidAttestationStatus[batchCid];
bool isSubmissionWindowOver = epochInfo[epochId].blocknumber + attestationSubmissionWindow <
block.number;
bool hasRequiredAttestations = maxAttestationsCount[batchCid] >= minAttestationsForConsensus;
bool hasValidRootHash = bytes32(maxAttestationFinalizedRootHash[batchCid]) != bytes32(0);
```

This would make the consensus validation logic easier to understand and maintain.

S9 Key State Variables Not Initialized in `initialize()`

Fixed

File(s) affected: `ProtocolState.sol`

Description: The `snapshotterState` and `dataMarketFactory` instances are not initialized in the `ProtocolState.initialize()` function. The contract will fail to operate correctly if functions are called before these variables are initialized.

Recommendation: Consider initializing these variables during the `initialize()` function.

S10 Remove Unused Struct Fields

Acknowledged

File(s) affected: `SnapshotterState.sol`

Description: The `LegacyVestingNodeInfo` struct is used in the `nodeIdToVestingInfo` mapping. However, after being set in the `burnNode()` function only the `tokensClaimed` and `lastClaim` fields are used again within the contract. However, the `initialClaim` and `tokensAfterInitialClaim` fields will be identical for all node IDs in the `nodeIdToVestignInfo` mapping, making them redundant.

Recommendation: Consider removing these redundant fields to reduce contract bloat and improve readability.

S11 Inconsistent Native Token Transfer Methods Used

Fixed

File(s) affected: `SnapshotterState.sol`

Description: When sending users rewards in `SnapshotterState.claimNodeTokens()` the contract mixes between using the `call` and `transfer` solidity keywords to transfer native tokens to the user.

Recommendation: Unless there is a good reason not to, consider using the same transfer method to increase consistency within the codebase.

S12 Members of `SnapshotterState.NodeInfo` Can Be Packed More Efficiently

Acknowledged

File(s) affected: `SnapshotterState.sol`

Description: Storage slots are always 32 bytes wide. Placing variables next to each other which are combined less than 32 bytes will pack them in the same storage slot, making operations more gas efficient.

Recommendation: Place the `bool` variables next to the `address` variable.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Files

- `315...95b ./contracts/DataMarket.sol`
- `d21...c33 ./contracts/DataMarketFactory.sol`
- `58b...955 ./contracts/ProtocolState.sol`
- `e2f...f54 ./contracts/SnapshotterState.sol`

Tests

- `e52...55f ./test/DataMarketFactory.js`
- `795...9a6 ./test/PowerloomNodes.js`
- `355...a73 ./test/ProtocolState.js`
- `859...001 ./test/TestUpgrades.js`

Test Suite Results

Run `npx hardhat test`

```
DataMarket Deployment
    ✔ should create a DataMarket
    ✔ should correctly set the data market count and addresses in ProtocolState
    ✔ Should release Event if data market is created

  PowerloomNodes
    Initialization
      ✔ Should set the right owner (69ms)
      ✔ Should initialize with correct values
      ✔ Should allow owner to update admins
      ✔ Should allow the owner to update the URI
      ✔ Should allow the owner to set mint start time
      ✔ Should allow the owner to set snapshotter address change cooldown
      ✔ Should allow the owner to pause and unpause the contract
      ✔ Should allow the owner to update the node price
      ✔ Should allow the owner to update the snapshotter token claim cooldown
    Transfer
      ✔ Should not allow safe transfers from or safe batch transfers
    Node Minting
      ✔ Should allow a user to create a node with sufficient power
      ✔ Should return excess ETH if more than enough power is sent on mint
      ✔ Should not allow creating a node with insufficient power
      ✔ Should allow node owner to burn their node and update node info correctly
      ✔ Should restrict the burning of a node under certain conditions
      ✔ Should not allow burning a node when the contract is paused
      ✔ Should update the node active status when an assigned node is burned
    Snapshotter Assignment
      ✔ Should allow an owner to assign a snapshotter to a node (42ms)
      ✔ Should allow bulk assignment of snapshotters to nodes by owner (38ms)
      ✔ Should allow an admin to assign snapshotters to nodes individually and in bulk
      ✔ Should return if a node is available or not
    Legacy Node Management
      ✔ Should allow the owner to update the name
      ✔ Should allow the owner to configure legacy nodes
      ✔ Should revert if initial claim percentage is too high
      ✔ Should revert if tokens sent on L1 is greater than or equal to node value
      ✔ Should revert if legacy node value is zero
      ✔ Should allow admin to mint legacy nodes
      ✔ Should fail to mint a legacy node if legacy nodes are not configured
      ✔ Should calculate vested legacy node tokens correctly
      ✔ Should calculate claimable legacy node tokens correctly
      ✔ Should send the correct initial claim when a legacy node is burned
      ✔ Should allow standard node holders to claim their node tokens
      ✔ Should allow legacy node holders to claim their node tokens
      ✔ Should prevent claiming of node tokens when the contract is paused
    Emergency Withdraw
      ✔ Should allow the owner to emergency withdraw power

  PowerloomProtocolState
    Deployment
      ✔ Should set the right owner
    Snapshotter State
      ✔ should bulk assign snapshotters to slots
    Data Market Protocol State
      ✔ Should set the right protocol state proxy address
      ✔ Should update protocol state in data market contract
    Release Epoch
      ✔ Should fail if epoch manager is not set
      ✔ Should set epoch manager
      ✔ Should fail if end epoch is less than start epoch
      ✔ Should fail if the epoch size is incorrect
      ✔ Should fail if the epoch is not continuous
      ✔ Should not revert if end epoch is greater than start epoch
      ✔ Should release Event if end epoch is greater than start epoch
    Submit Snapshots
      ✔ Should return empty string for empty cids in snapshot
      ✔ Should set the project first epoch id on batch submission
      ✔ Batch submission should fail if submitted again
      ✔ Should fail if the project ids and snapshot cids length mismatch
      ✔ Should store batch attestations successfully
      ✔ Should finalize batch on enough batch attestations
      ✔ Should correctly handle consensus for attestations
      ✔ Should force consensus for attestations
      ✔ Should fail to submit batch attestation if batch cid is not submitted
```

```
        ✔ Should properly handle delayed submissions
        ✔ Should properly handle delayed attestations
        ✔ Should properly handle divergent validators
        ✔ Should correctly force complete consensus attestation with divergent validators
        ✔ Should trigger resubmission if validators are divergent
        ✔ Should store batchId to projectIds mapping
        ✔ Should store the correct consensus data for different types of finalization
        ✔ Should handle batch submissions with partial duplicates
        ✔ Should store epochId to batchIds mapping
        ✔ Should store the correct batchCid after sequencer submission
        ✔ Should end batch submissions
      Rewards
        ✔ Should return correct rewards sum
        ✔ Should store rewards successfully
        ✔ Should successfully claim rewards
        ✔ Should successfully claim rewards from multiple data markets
      Protocol State Getters/Setters
        ✔ Should toggle data market active status
        ✔ Should fail to update addresses if role is invalid
        ✔ Should fail to update addresses if array length is invalid
        ✔ Should get total snapshotter count correctly
        ✔ Should succesfully remove addresses
        ✔ Should update the protocol state contract addresses
        ✔ Should successfully update the data market storage mappings
      Data Market Getters/Setters
        ✔ Should update data market roles
        ✔ Should update day size
        ✔ Should get and update data market state
        ✔ Should get and set epoch related data from the data market contract
        ✔ Should get and set epoch related data from the data market contract with blocknumber for
epochId
        ✔ Should get and set sequencer data from the data market contract
        ✔ Should get and set validator data from the data market contract
        ✔ Should get the reward related data from the data market contract
        ✔ Should successfully get and update slot related data
        ✔ Should successfully get and update reward data
      Access Control
        ✔ Should successfully limit access to onlyOwner modified functions
        ✔ Should successfully limit access to the DataMarket onlySequencer and onlyValidator
functions
        ✔ Should successfully limit access to the DataMarket onlyOwnerOrigin modified functions
        ✔ Should successfully limit access to the DataMarket isActive modified functions
      Emergency Withdraw
        ✔ Should successfully emergency withdraw rewards

    PowerloomProtocolState Upgrade
      ✔ should upgrade to PowerloomProtocolStateV2 and keep the same address
      ✔ should return the correct string from newFunctionality after upgrade
      ✔ should correctly set and get values in newMapping after upgrade
      ✔ should break if you try to deploy incompatible memory layout in an upgrade

    PowerloomProtocolState
      Deployment
        ✔ Should set the right owner


    100 passing (6s)
```

Fix Review Update

```
  DataMarket Deployment
      ✔ should create a DataMarket
      ✔ should correctly set the data market count and addresses in ProtocolState
      ✔ Should release Event if data market is created
      ✔ Should deploy an implementation contract on creation

    PowerloomNodes
    Initialization
      ✔ Should set the right owner (94ms)
      ✔ Should initialize with correct values
      ✔ Should allow owner to update admins
      ✔ Should allow the owner to update the URI
      ✔ Should allow the owner to set max supply
      ✔ Should allow the owner to set mint start time
      ✔ Should allow the owner to set snapshotter address change cooldown
      ✔ Should allow the owner to pause and unpause the contract
      ✔ Should allow the owner to update the node price
      ✔ Should allow the owner to update the snapshotter token claim cooldown
    Transfer
      ✔ Should not allow safe transfers from or safe batch transfers
    Node Minting
      ✔ Should allow a user to create a node with sufficient power
      ✔ Should return excess ETH if more than enough power is sent on mint
      ✔ Should not allow creating a node with insufficient power
      ✔ Should allow node owner to burn their node and update node info correctly
      ✔ Should restrict the burning of a node under certain conditions
      ✔ Should not allow burning a node when the contract is paused
      ✔ Should update the node active status when an assigned node is burned
    Snapshotter Assignment
      ✔ Should allow an owner to assign a snapshotter to a node (74ms)
      ✔ Should allow bulk assignment of snapshotters to nodes by owner (101ms)
      ✔ Should allow an admin to assign snapshotters to nodes individually and in bulk (81ms)
      ✔ Should return if a node is available or not
    Legacy Node Management
      ✔ Should allow the owner to update the name
      ✔ Should allow the owner to configure legacy nodes
      ✔ Should revert if initial claim percentage is too high
      ✔ Should revert if tokens sent on L1 is greater than or equal to node value
      ✔ Should revert if legacy node value is zero
      ✔ Should allow admin to mint legacy nodes (41ms)
      ✔ Should fail to mint a legacy node if legacy nodes are not configured
      ✔ Should calculate vested legacy node tokens correctly
      ✔ Should calculate claimable legacy node tokens correctly
      ✔ Should send the correct initial claim when a legacy node is burned
      ✔ Should allow standard node holders to claim their node tokens (43ms)
      ✔ Should allow legacy node holders to claim their node tokens (57ms)
      ✔ Should prevent claiming of node tokens when the contract is paused
    Emergency Withdraw
      ✔ Should allow the owner to emergency withdraw power

  PowerloomProtocolState
    Deployment
      ✔ Should set the right owner
    Data Market Factory
      ✔ Should set the right protocol state proxy address
      ✔ Should create a new data market and verify the initial state (48ms)
    Snapshotter State
      ✔ should bulk assign snapshotters to slots (71ms)
    Data Market Protocol State
      ✔ Should set the right protocol state proxy address
    Release Epoch
      ✔ Should fail if epoch manager is not set
      ✔ Should set epoch manager
      ✔ Should test epoch release with epoch size 1 and USE_BLOCK_NUMBER_AS_EPOCH_ID true (47ms)
      ✔ Should simulate force release epochs, and simulate day increment (347ms)
      ✔ Should fail if end epoch is less than start epoch
      ✔ Should fail if the epoch size is incorrect
      ✔ Should fail if the epoch is not continuous (45ms)
      ✔ Should not revert if end epoch is greater than start epoch
      ✔ Should release Event if end epoch is greater than start epoch
    Submit Snapshots
      ✔ Should return empty string for empty cids in snapshot
      ✔ Should set the project first epoch id on batch submission (38ms)
      ✔ Batch submission should fail if submitted again (78ms)
      ✔ Should fail if the project ids and snapshot cids length mismatch (45ms)
      ✔ Should store batch attestations successfully (50ms)
      ✔ Should finalize batch on enough batch attestations (173ms)
      ✔ Should correctly handle consensus for attestations (154ms)
      ✔ Should force consensus for attestations (146ms)
      ✔ Should fail to submit batch attestation if batch cid is not submitted
      ✔ Should properly handle delayed submissions
      ✔ Should properly handle delayed attestations (74ms)
      ✔ Should properly handle divergent validators (173ms)
      ✔ Should correctly force complete consensus attestation with divergent validators (131ms)
      ✔ Should trigger resubmission if validators are divergent (119ms)
      ✔ Should store batchId to projectIds mapping (55ms)
      ✔ Should store the correct consensus data for different types of finalization (117ms)
      ✔ Should handle batch submissions with partial duplicates (93ms)
      ✔ Should store epochId to batchIds mapping (49ms)
      ✔ Should store the correct batchCid after sequencer submission (48ms)
      ✔ Should end batch submissions
    Rewards
      ✔ Should return correct rewards sum (104ms)
      ✔ Should update eligible nodes for day atomically (113ms)
      ✔ Should not assign rewards to non-existent slots or slots that have already received
rewards (67ms)
      ✔ Should store rewards successfully (64ms)
      ✔ Should successfully claim rewards (69ms)
      ✔ Should successfully claim rewards from multiple data markets (127ms)
    Protocol State Getters/Setters
      ✔ Should toggle data market active status
      ✔ Should fail to update addresses if role is invalid
      ✔ Should fail to update addresses if array length is invalid
      ✔ Should get total snapshotter count correctly (60ms)
      ✔ Should succesfully remove addresses
      ✔ Should update the protocol state contract addresses
      ✔ Should successfully update the data market storage mappings
    Data Market Getters/Setters
      ✔ Should update data market roles
      ✔ Should get and update data market state
      ✔ Should get and set epoch related data from the data market contract (79ms)
      ✔ Should get and set epoch related data from the data market contract with blocknumber for
epochId (91ms)
      ✔ Should get and set sequencer data from the data market contract
      ✔ Should get and set validator data from the data market contract
      ✔ Should get the reward related data from the data market contract
      ✔ Should successfully get and update slot related data (88ms)
      ✔ Should successfully get and update reward data (45ms)
    Access Control
      ✔ Should successfully limit access to onlyOwner modified functions
      ✔ Should successfully limit access to the DataMarket onlySequencer and onlyValidator
functions (53ms)
      ✔ Should successfully limit access to the DataMarket onlyOwnerOrigin modified functions
(97ms)
      ✔ Should successfully limit access to the DataMarket isActive modified functions (45ms)
    Emergency Withdraw
      ✔ Should successfully emergency withdraw rewards

  PowerloomProtocolState Upgrade
    ✔ should upgrade to PowerloomProtocolStateV2 and keep the same address (47ms)
    ✔ should return the correct string from newFunctionality after upgrade
    ✔ should correctly set and get values in newMapping after upgrade
    ✔ should break if you try to deploy incompatible memory layout in an upgrade

  Data Market Upgrade
    ✔ should upgrade to TestPowerloomDataMarket and keep the same address
    ✔ should return the correct string from newFunctionality after upgrade
    ✔ should verify that the data market contract state is the same (190ms)

  Snapshotter State Upgrade
    ✔ should upgrade to PowerloomNodesUpgrade and keep the same address
    ✔ should return the correct string from newFunctionality after upgrade
    ✔ should verify that the contract state is preserved after upgrade (158ms)
```

```
111 passing (14s)
```

Code Coverage

Run `npx hardhat coverage`

```
-----------------------------|----------|----------|----------|----------|----------------|
File                         |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-----------------------------|----------|----------|----------|----------|----------------|
 contracts/                  |    97.62 |    84.97 |    97.13 |    96.88 |                |
  DataMarket.sol             |    99.19 |    88.96 |      100 |    97.91 |... 21,977,1050 |
  DataMarketFactory.sol      |      100 |       50 |      100 |      100 |                |
  ProtocolState.sol          |      100 |    85.71 |      100 |      100 |                |
  SnapshotterState.sol       |    94.48 |    82.26 |    88.64 |    93.78 |... 394,395,396 |
 contracts/TestArtifacts/    |    28.57 |        0 |    28.57 |    33.33 |                |
  ProtocolStateV2.sol        |      100 |      100 |      100 |      100 |                |
  ProtocolStateV3Broken.sol  |        0 |        0 |        0 |        0 |... 25,32,33,37 |
-----------------------------|----------|----------|----------|----------|----------------|
All files                    |     96.5 |     84.1 |    94.48 |    95.84 |                |
-----------------------------|----------|----------|----------|----------|----------------|
```

Fix Review Update

```
-------------------------|----------|----------|----------|----------|----------------|
File                     |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-------------------------|----------|----------|----------|----------|----------------|
 contracts/              |      100 |    80.26 |      100 |      100 |                |
  DataMarket.sol         |      100 |    77.83 |      100 |      100 |                |
  DataMarketFactory.sol  |      100 |       50 |      100 |      100 |                |
  ProtocolState.sol      |      100 |    78.79 |      100 |      100 |                |
  SnapshotterState.sol   |      100 |    83.87 |      100 |      100 |                |
-------------------------|----------|----------|----------|----------|----------------|
All files                |      100 |    80.26 |      100 |      100 |                |
-------------------------|----------|----------|----------|----------|----------------|
```

Changelog

- 2025-01-21 - Initial report
- 2025-02-07 - Fix review

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Quantstamp