Polygon

Aggregation Layer

Security Assessment Report

Version: 2.0

December, 2024

Contents

Introduction 2
- Disclaimer 2
- Document Structure 2
- Overview 2

Security Assessment Summary 3
- Scope 3
- Approach 3
- Coverage Limitations 4
- Findings Summary 4

Detailed Findings 5

Summary of Findings 6
- Proofs Are Not Settled To L1 7
- Stale Nodes 8
- Error During Packing Causes Desync Between Local and Onchain State 9
- Lack of Unique Nonce in `hash()` Function May Result In Duplicate Hashes and Replay Attacks 10
- Lack Of Distinction Between Leaf And Node Hashes 11
- No Checks For Duplicate Values During `insert()` 12
- Unauthenticated AggLayer RPC 13
- Unchecked Panic With `unwrap()` 14
- Proof Aggregation Does Not Occur 15
- Lack Of Max Depth Checks 16
- Pessimistic Rollups Can Call `onSequenceBatches()` 17
- Vulnerable Crate Dependencies 18
- Insufficient `default_rpc_confirmations` 19
- `BlockClock` Does Not Account For Reorgs 20
- Repeated Transactions Continue To Attempt Settlement 21
- Unwrapping With `unwrap_or()` May Result In Silent Failures 22
- Lack Of Atomicity 23
- Fixed Tree Depth 24
- `rollupContract` Value Type Is Incompatible With Pessimistic Rollups 25
- No Zero Check On `programVKey` 26
- Inconsistent Timekeeping Due to Skipped Ethereum L1 Blocks 27
- `TimeClock` Starts From Epoch Zero 28
- `SignedTx::hash()` does not contain `rollup_id` 29
- `CertificateOrchestrator::received_certificates` Will Grow Unbounded 30
- Miscellaneous General Comments 31

A Test Suite 34

B Vulnerability Severity Classification 37

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Polygon smart contracts and Rust implementation of the AggLayer. The review focused solely on the security aspects of the Solidity and Rust implementation of the contracts and off-chain code, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Polygon components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Polygon components in scope.

Overview

The Aggregation Layer ("AggLayer"), serves as a decentralized protocol to transform the fragmented blockchain landscape. Acting as a unifying force, it unites disparate L1 and L2 chains, fortified with ZK-security, into a cohesive network that operates akin to a single chain.

The AggLayer operates on two fundamental principles: aggregating ZK proofs from interconnected chains and ensuring the safety of near-instant atomic cross-chain transactions.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the zkevm-contracts and agglayer repositories.

The scope of the offchain portion of this time-boxed review was strictly limited to files at commit d9d3388.

The scope of the smart contract portion of this time-boxed review was strictly limited to the specific changes in the c4eda49..2de7a151 diff and the following contracts:

1. `PolygonRollupManager.sol`
2. `PolygonZkEVMGlobalExitRootV2.sol`
3. `PolygonPessimisticConsensus.sol`
4. `PolygonConsensusBase.sol`
5. `PolygonRollupBaseEtrog.sol`

Note: third party libraries and dependencies, such as OpenZeppelin, were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the solution. This includes their interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout) and overall Rust implementation.

The manual review process of the on-chain portion of this assessment focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

The off-chain portion of this assessment focused on identifying known Rust anti-patterns and attack vectors. These include, but are not limited to, the following vectors: error handling and wrapping, panicking macros, arithmetic errors, UTF-8 strings handling, index out of bounds and resource exhaustion.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Aderyn: `https://github.com/Cyfrin/aderyn`
- cargo audit: `https://crates.io/crates/cargo-audit`
- cargo deny: `https://github.com/EmbarkStudios/cargo-deny`
- cargo tarpaulin: `https://crates.io/crates/cargo-tarpaulin`

- cargo geiger: `https://github.com/rust-secure-code/cargo-geiger`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 25 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 3 issues.
- Medium: 5 issues.
- Low: 7 issues.
- Informational: 9 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Polygon components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
AGG-01	Proofs Are Not Settled To L1	Critical	Resolved
AGG-02	Stale Nodes	High	Closed
AGG-03	Error During Packing Causes Desync Between Local and Onchain State	High	Resolved
AGG-04	Lack of Unique Nonce in `hash()` Function May Result In Duplicate Hashes and Replay Attacks	High	Closed
AGG-05	Lack Of Distinction Between Leaf And Node Hashes	Medium	Closed
AGG-06	No Checks For Duplicate Values During `insert()`	Medium	Closed
AGG-07	Unauthenticated AggLayer RPC	Medium	Closed
AGG-08	Unchecked Panic With `unwrap()`	Medium	Resolved
AGG-09	Proof Aggregation Does Not Occur	Medium	Closed
AGG-10	Lack Of Max Depth Checks	Low	Resolved
AGG-11	Pessimistic Rollups Can Call `onSequenceBatches()`	Low	Resolved
AGG-12	Vulnerable Crate Dependencies	Low	Closed
AGG-13	Insufficient `default_rpc_confirmations`	Low	Closed
AGG-14	`BlockClock` Does Not Account For Reorgs	Low	Closed
AGG-15	Repeated Transactions Continue To Attempt Settlement	Low	Closed
AGG-16	Unwrapping With `unwrap_or()` May Result In Silent Failures	Low	Resolved
AGG-17	Lack Of Atomicity	Informational	Closed
AGG-18	Fixed Tree Depth	Informational	Closed
AGG-19	`rollupContract` Value Type Is Incompatible With Pessimistic Rollups	Informational	Resolved
AGG-20	No Zero Check On `programVKey`	Informational	Closed
AGG-21	Inconsistent Timekeeping Due to Skipped Ethereum L1 Blocks	Informational	Closed
AGG-22	`TimeClock` Starts From Epoch Zero	Informational	Closed
AGG-23	`SignedTx::hash()` does not contain `rollup_id`	Informational	Closed
AGG-24	`CertificateOrchestrator::received_certificates` Will Grow Unbounded	Informational	Resolved
AGG-25	Miscellaneous General Comments	Informational	Closed

AGG-01	Proofs Are Not Settled To L1		
Asset	`agglayer-aggregator-notifier/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The proofs received by the AggLayer client are not currently published on Ethereum L1, which serves as the settlement layer for Polygon AggLayer. This lack of publication results in the AggLayer client state becoming increasingly out of sync with the blockchain state. Consequently, the system fails to function effectively as a layer 2 solution since no critical data is being written to Ethereum L1.

While there is a mechanism in place to settle transactions on Ethereum L1, it has not been fully integrated into the protocol, with integration only occurring in some test scenarios. This oversight further exacerbates AGG-03, undermining the reliability and integrity of the AggLayer client as a layer 2 solution. Without proper data settlement on Ethereum L1, the system risks operational failures and could face significant challenges in maintaining consistency and trust with users.

Recommendations

Add the settlement of proof to the evaluation cycle of the AggLayer client.

Resolution

The finding has been resolved in PR-348.

AGG-02	Stale Nodes		
Asset	`pessimistic-proof/src/utils/smt.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `insert_helper()` function inserts and updates intermediate nodes based on the data that was entered, but it does not remove old hashes. As a result, it may be possible to reference old values by simply referencing old hashes. This is particularly dangerous when updating values, as it could be possible to create a proof for an old value.

Retaining old hashes and nodes may lead to increased memory consumption over time, especially if many insertions or updates are performed. Each new node insertion results in storing a new hash and node pair, while the old hashes are left intact, potentially leading to a growing amount of unused or "stale" data.

If the application relies on ensuring that only the latest state is valid and accessible, having stale hashes could potentially lead to security issues. For example, if an attacker can access or reference these stale nodes, it could lead to unintended behavior, such as replay attacks or unauthorized access to outdated data.

Referencing old hashes could compromise the integrity of the tree. For instance, if the application logic allows navigation to nodes by hash directly, it might end up using an outdated node instead of the latest version.

Recommendations

Modify `insert_helper()` to ensure that when a node is updated, any associated old hashes are identified and removed from the tree. This can be done by keeping track of the hashes being replaced and purging them after the update.

Resolution

The development team has acknowledged this finding with the following comment:

> *"This SMT implementation is meant to be a simple prototype that will be optimized in the future, so I think we can take care of removing the old hashes then."*

AGG-03	Error During Packing Causes Desync Between Local and Onchain State		
Asset	agglayer-certificate-orchestrator/src/lib.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When a certificate is received from the network, the `certify()` function is triggered to update the chain's local `global_state` in `CertificateOrchestrator`, reflecting the new state changes included in the certificate. After certification, the certificate is then packed using the `pack()` function, which, when fully implemented, aggregates proofs and settles them on Ethereum L1.

However, if the packing process fails (i.e., no settlement occurs on L1), the updates made to the `global_state` are not reverted. This creates a discrepancy between the local `global_state` and the actual settled state on Ethereum L1.

Since the local `global_state` is used by `certify()` to process future certificates, this inconsistency can cause the certification of subsequent incoming certificates to fail. In turn, this could result in a Denial of Service (DoS) condition, potentially requiring manual intervention to correct the issue and restore the canonical state.

Recommendations

Revert changes to `self.global_state` in the case that packing fails. This would ensure local state remains consistent with the onchain state.

Resolution

The finding has been resolved in commit 6e17c8e by moving the logic from the `CertificateOrchestrator` to a `NetworkTask` that is responsible for dealing with its own state update.

AGG-04	Lack of Unique Nonce in `hash()` Function May Result In Duplicate Hashes and Replay Attacks		
Asset	`pessimistic-proof/src/bridge_exit.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

In the `hash()` function on line [**102**] of `pessimistic-proof/src/bridge_exit.rs`, there is no unique nonce implemented to ensure that each hash is distinct.

The current fields used in the hash computation, namely `leaf_type`, `origin_network`, `origin_token_address`, `dest_network`, `dest_address`, `amount`, and `metadata`, are insufficient to prevent the possibility of generating duplicate hashes.

As a result, it may be feasible for an attacker to replay the same hash to initiate a bridge exit multiple times with the same parameters, leading to unauthorized exits.

Recommendations

Implement a unique nonce as part of the hash computation. By incorporating a nonce, each hash will be guaranteed to be unique, effectively preventing potential replay attacks and ensuring the integrity of bridge exit transactions.

Resolution

The finding has been closed as false-positive with the following comment from the development team:

> *"The only unique thing "nonce" will be what we use as nullifier, which will be always a set of: originNetwork and originTokenAddress. This identifies uniquely a leaf.*
>
> *All the rest of parameters can be the same, e.g it's possible for a user to send 1 ether twice.*
>
> *When exiting, you have to provide a merkle prof against a leaf (an index). This index gets nullified each time a claim happens (so it's not possible for user B to reuse user's A hash to exit user A twice).".*

AGG-05	Lack Of Distinction Between Leaf And Node Hashes		
Asset	`pessimistic-proof/src/utils/smt.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Current implementation does not adequately distinguish between leaf nodes and intermediate nodes (which contain the hash of two child nodes).

Note, due to a fixed tree depth, potential security issues of extending trees are mitigated, however, this issue could be compounded by the AGG-04 finding.

The `insert_helper()` function currently inserts and updates intermediate nodes based on new data without removing old hashes. As a result, there is a possibility, albeit unlikely, that traversing the tree could inadvertently reference outdated hashes, leading to incorrect node values.

This scenario could arise if, during tree traversal, an old hash is referenced, causing the process to return to an outdated node further up the tree. Ultimately, this could result in retrieving a node value rather than a leaf value due to mismatched tree depth.

Recommendations

Implement a type flag or metadata on each node indicating whether it is a leaf node or an intermediate node. This metadata can either be incorporated into the node structure or maintained as a separate field.

Additionally, during the verification of a Merkle proof, check the type of each node along the proof path.

Resolution

The development team has acknowledged this finding with the following comment:

> *"It would require a corrupt tree, or hash collisions, or something else to go drastically wrong, so the result would be corrupted in any case, with or without this scenario of pointing to a stale node. In any case, I think we'll have to implement the distinction between leaf and node hashes when we optimize the SMT (e.g. with pruning) so this will be fixed at some point. This issue doesn't apply to our current design.".*

AGG-06	No Checks For Duplicate Values During `insert()`		
Asset	`pessimistic-proof/src/utils/smt.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `insert_helper()` function does not account for hash collisions on line [197] when inserting a new key-value into the tree. Duplicate hashes, although highly unlikely, may result in unexpected behaviour as during the `insert()` into `HashMap`, the existing value will be overwritten (i.e., the newly added node will be referenced by the old hash).

This is especially problematic for operations like inclusion proofs, where traversing the tree could lead to incorrect paths and values taken.

Recommendations

Implement a mechanism to detect hash collisions before inserting a new key-value pair into the HashMap. If a collision is detected (i.e., if the hash already exists in the map), take appropriate steps to resolve it.

Resolution

The development team has acknowledged this finding with the following comment:

> *"This issue relies on finding a Keccak hash collision. Our code works under the assumption that Keccak is resistant to hash collisions and I don't think it is necessary to add extra checks for that".*

AGG-07	Unauthenticated AggLayer RPC		
Asset	`agglayer-node/src/rpc/mod.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The RPC server started in `AgglayerImpl::start()` has no authentication requirements.

As such, if an attacker can reach this RPC endpoint, they can trivially send unauthorized requests. For example, an attacker could fill up the `max_connections` and trigger Denial-of-Services (DoS) conditions.

Recommendations

Ensure the RPC server is not publicly exposed. Alternatively, consider adding authentication to the RPC server.

Resolution

The development team has acknowledged this finding with the following comment:

> *"That's something that we didn't plan to do for the upcoming version, and we need to do some design on how we want to manage the access to the RPC. Currently, we're operating both AggLayer node and AggSender. This issue can stay open, but it'll be either fix later or not done based on the design decision that we'll make."*

AGG-08	Unchecked Panic With `unwrap()`		
Asset	`agglayer-clock/src/block.rs, agglayer-node/build.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

It was observed that multiple instances of the `unwrap()` method are used throughout the code. This function, while convenient for handling `Option` and `Result` types, can lead to unhandled panics if called on an `Err` variant without sufficient validation.

Panics triggered by `unwrap()` can terminate the program abruptly, potentially leading to unexpected behaviour or denial of service, particularly in production systems where stability and fault tolerance are critical.

The particular instances noted were (note, the list below should not be considered exhaustive):

1. line [**25**] and line [**33**] of `agglayer-node/build.rs` relating to accessing file systems. These requests can result in errors due to lack of permissions, file system being full or the file not existing.
2. line [**201**] and line [**202**] of `agglayer-clock/src/block.rs` relating to accessing fields of a block. If a block is pending, then accessing `block.number` will return an error. Instances that would cause `block.hash` to return an error were not immediately clear, but it is prudent to defensively prevent such issues from arising.

As these issues would be unlikely to occur in an active, properly configured system, this finding has been assigned a low likelihood.

Recommendations

Handle `Result` types more robustly by using safer alternatives, such as:

- Pattern matching: Explicitly handle `Ok/Err` cases using match statements, ensuring all possible cases are covered.
- `if let` or `while let` constructs: These provide more concise handling of successful cases, with fallback behaviour in case of errors or missing values.
- Custom error handling: Return relevant error messages or propagate errors using the `?` operator to gracefully handle failure scenarios, allowing errors to propagate up to higher levels of the application.

Resolution

The finding has been resolved in commit ce2cadb and PR-361.

AGG-09	Proof Aggregation Does Not Occur		
Asset	`agglayer-aggregator-notifier/src/lib.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Currently, Pessimistic Proofs are intended to be settled individually as different transactions on Ethereum L1, instead of aggregating the proofs together into a batch to be settled.

The result of this will be a higher operating cost for the AggLayer client and reduced throughput if demand for settlement is high, given Ethereum's sequential transaction nature.

This issue has been rated as low impact, as proofs being aggregated is an optimization and it does not affect the core functionality of the system.

The testing team thought it was important to highlight this deviation from the specification as implementation of proof aggregation is non-trivial and a common area for vulnerabilities to appear.

Recommendations

Integrate aggregation of proofs when settling to Ethereum L1 as demand dictates.

It is recommended to have the aggregation implementation thoroughly reviewed to prevent further vulnerabilities arising from the updated code.

Resolution

The development team has acknowledged this finding with the following comment:

> *"The proof aggregation isn't part of the 0.2.0 release, this feature will be launched in a further version as we're still designing how to do it."*

AGG-10	Lack Of Max Depth Checks		
Asset	`pessimistic-proof/src/utils/smt.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In `insert_helper()` there are no checks to ensure that `depth` is not greater than or equal to `DEPTH`, which could result in an underflow when calculating the index `[DEPTH - depth - 1]`.

This underflow would occur if `depth` is equal to or greater than `DEPTH`, causing the index to reference an invalid memory location.

Recommendations

Before performing the index calculation, validate that `depth` is strictly less than `DEPTH`. If `depth` is out of bounds, return an error or handle the situation to prevent underflow.

Resolution

The finding has been resolved in PR-325 and PR-361.

AGG-11	Pessimistic Rollups Can Call `onSequenceBatches()`		
Asset	`PolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is no check in `onSequenceBatches()` to prevent it being called by pessimistic rollups.

If a pessimistic rollup were to call this function to sequence batches, it could modify `totalSequencedBatches`, decreasing the verification reward per batch. It could also block itself from being upgraded through `updateRollupByRollupAdmin()` owing to the check on line [**662**].

This issue is significantly mitigated by the pessimistic rollup contracts not calling `onSequenceBatches()`. However, `updateRollupByRollupAdmin()` can update a rollup to any contract, including one that calls this function. Similarly, if a pessimistic rollup contract were to contain a security vulnerability that allowed the contract to call out, `onSequenceBatches()` has no check against it. As `verifyBatchesTrustedAggregator()` and `verifyPessimisticTrustedAggregator()` both check `VerifierType`, it might be difficult to remedy the issue in this case.

Recommendations

Consider checking the `VerifierType` in `onSequenceBatches()`.

Resolution

The finding has been addressed in the latest versions of PP (such as `v9.0.0-rc.1-pp` and above) and can be observed already implemented in commit f0ba99b.

AGG-12	Vulnerable Crate Dependencies		
Asset	`agglayer-aggregator-notifier, agglayer-gcp-kms`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Rust advisory lists two crates that exist in the dependency tree that have vulnerabilities which may cause unexpected panics:

- quinn-proto version 0.11.6: it is advised to use 0.11.7 or greater.

```
quinn-proto 0.11.6
└── quinn 0.11.3
└── reqwest 0.12.5
├── twirp-rs 0.3.0
│ └── sp1-sdk 1.1.1
│ ├── pessimistic-proof-test-suite 0.1.0
│ └── agglayer-aggregator-notifier 0.1.0
│ └── agglayer-node 0.1.0
│ └── agglayer 0.1.0
├── sp1-sdk 1.1.1
└── reqwest-middleware 0.3.3
└── sp1-sdk 1.1.1
```

- tonic version 0.9.2: this is advised to use 0.9.3 or greater.

```
tonic 0.9.2
├── gcloud-sdk 0.20.7
│ └── ethers-gcp-kms-signer 0.1.5
│ └── agglayer-gcp-kms 0.1.0
│ └── agglayer-signer 0.1.0
│ └── agglayer-node 0.1.0
│ └── agglayer 0.1.0
└── ethers-gcp-kms-signer 0.1.5
```

Recommendations

It is recommended to update vulnerable crate versions to their latest, stable versions to avoid potential unhandled panic situations.

Resolution

The development team has acknowledged this finding with the following comment:

> *"Vulnerabilities on deps are tracked using the audit workflow that is part of every CI status checks. Critical error on these checks makes the PR status failing, warnings are treated if necessary.".*

AGG-13	Insufficient `default_rpc_confirmations`		
Asset	`agglayer-config/src/outbound.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Currently, the settlement of a proof in `Kernel::settle()` is successful if it has 1 confirmation. However, if a block reorg with depth > 1 occurs, a settlement may not be included in the final chain.

In this scenario, the AggLayer will have marked the transaction as successful even though it was not settled. This results in a difference between the local state and onchain state.

agglayer-config/src/outbound.rs

```rust
const fn default_rpc_confirmations() -> usize {
    1
}
```

Recommendations

Consider using a higher value in `default_rpc_confirmations()`.

Resolution

The development team has acknowledged this finding with the following comment:

> *"This is by design the expectation of the v0.1 FEP, it lets the burden of tracking this to the chain itself instead of the AggLayer. The AggLayer is exposing an RPC to check the evolution of the tx on L1.".*

AGG-14	`BlockClock` Does Not Account For Reorgs		
Asset	`agglayer-clock/src/block.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `BlockClock` implementation subscribes to incoming blocks on Ethereum L1 using an external RPC to track time. However, the current implementation does not account for block reorganizations (reorgs), which can occur when a new block builds on top of a previous block's parent (n-1) rather than the latest block (n).

In this scenario, `stream.next` will register the new block and increment `self.block_height`, even though the actual on-chain block height remains unchanged. As a result, the local `self.block_height` may become out of sync, reflecting a higher value than the actual blockchain height.

When the AggLayer node is restarted, `self.block_height` is reset to the actual, lower, onchain block height. This discrepancy may cause an `EpochEnded` event to be emitted twice for the same epoch, leading to potential timing and logic issues in downstream systems or contracts relying on accurate epoch tracking.

Recommendations

Ensure the above comments are understood and consider changes if deemed applicable.

Resolution

The development team has acknowledged this finding and created a ticket internally to address this in further versions.

AGG-15	Repeated Transactions Continue To Attempt Settlement		
Asset	`agglayer-node/src/kernel/mod.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

During the settlement of a transaction to Ethereum L1, a check is performed to determine if the transaction has already been settled. However, if a duplicate transaction is detected, only a warning event log is emitted, and the settlement process continues, resulting in the repeated submission of the same transaction.

This approach is inefficient as repeating an already settled transaction wastes gas and can cause delays for other pending transactions. Additionally, if the transaction reverts upon repetition, it will continue to incur gas costs for each attempt, leading to unnecessary expenditure up to the `max_retries` limit. This not only strains resources but also impacts overall network performance.

Recommendations

Implement logic to check that if a transaction has been settled already, then the settlement process should return early instead of repeating settlement.

Resolution

The development team has acknowledged this finding with the following comment:

> *"This is linked to the FEP path of v0.1 and this warning was more of a safety checks to define if this happens at any moment. We don't plan to work on improving the v0.1 FEP path for now, and product may improve to the point where we just don't provide this feature anymore".*

AGG-16	Unwrapping With `unwrap_or()` May Result In Silent Failures		
Asset	`pessimistic-proof/src/local_exit_tree/data.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

`get()` function on line [98] will return an empty hash even for invalid indices:

```
pub fn get(&self, height: usize, index: usize) -> H::Digest {
    *self.layers[height].get(index).unwrap_or(&self.empty_hash_at_height[height])
}
```

The use of `unwrap_or()` in the above example means that if `self.layers[height].get(index)` returns `None`, it will default to `&self.empty_hash_at_height[height]`, which could mask errors or unexpected conditions where an index should exist, but does not, or when accessing indices out of bounds.

Recommendations

If `get()` should never return `None`, consider using `get().unwrap_or_else(|| ...)` where the closure could log the error or handle it in a way that does not silently proceed with potentially incorrect data.

Resolution

The finding has been resolved in PR-325 and PR-361 by introducing bound checks.

AGG-17	Lack Of Atomicity
Asset	`pessimistic-proof/src/utils/smt.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `insert_helper()` function, responsible for inserting nodes into the tree, is not atomic.

If the insertion process fails at any intermediate point—such as due to an unexpected error, panic, or system outage—the tree could be left in an inconsistent or partially updated state. This could compromise the integrity of the entire tree structure.

Recommendations

Consider making the `insert_helper()` function atomic. This can be done by ensuring that all changes made during the insertion process are either fully applied or fully rolled back in the event of an error.

This could be achieved with the following:

1. Implement a transactional mechanism where changes are staged in a temporary structure and only committed to the tree once the entire insertion process is successfully completed.
2. If an error occurs during the insertion, roll back any changes made up to that point to maintain the tree's consistency.

Resolution

The development team has acknowledged this finding with the following comment:

> *"While atomicity would certainly be nice to have in this function, I don't think it's especially urgent to implement. I think we can fix this in a later version."*

AGG-18	Fixed Tree Depth
Asset	`pessimistic-proof/src/local_exit_tree/data.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `add_leaf()` function assumes the fixed tree depth `TREE_DEPTH`. If the tree needs to grow beyond the dept, the function will fail due to the `assert_eq!` check ensuring that the number of leaves does not exceed `2 ^ TREE_DEPTH`.

Note, the `TREE_DEPTH` is currently hardcoded to `32`, meaning the max number of leaves that would fit in the tree is `2 ^ 32`, consider if this is sufficient to scale in the future.

Recommendations

Consider if max number of leaves (`2 ^ 32`) is sufficient to scale in the future.

Resolution

The development team has acknowledged the finding with the following comment:

> *"In the medium-term,* `2 ^ 32` *leaves is good enough. We can consider ways to increase this limit in future versions.".*

AGG-19	`rollupContract` Value Type Is Incompatible With Pessimistic Rollups
Asset	`PolygonRollupManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Within the new struct `RollupDataReturnV2` of `PolygonRollupManager`, the variable `rollupContract` is of type `IPolygonRollupBase`. However, `IPolygonPessimisticConsensus` does not inherit from this interface and so contracts of this type are not natively compatible with this variable.

Whilst the struct `RollupDataReturn` also has this issue, it is understood that this older struct is preserved for legacy compatibility.

Recommendations

Consider changing the type of `rollupContract` to `IPolygonConsensusBase` and making `IPolygonPessimisticConsensus` inherit from that interface. Or, if that particular configuration is not desired, consider an `address` type to maximise future compatibility.

Resolution

The finding has been resolved addressed in commit d05367e, the `rollupContract` was altered to become an `address` type to ensure future compatibility.

AGG-20	No Zero Check On `programVKey`
Asset	`PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The function `addNewRollupType()` of `PolygonRollupManager` allows a value of zero for its parameter `programVKey` when adding a new pessimistic rollup type. This would not be a valid value.

Recommendations

Consider adding a zero check on this parameter when the `rollupVerifierType` is `Pessimistic`.

Resolution

The development team has acknowledged this finding with the following comment:

> *"No extra verification has been done on the `verificationKey` parameter since it does not add any extra safety measure. `verificationKey` represents the summary of the circuit that is going to be verified, a hash of the circuit. Hence, zero value should be treated as any other value".*

AGG-21	Inconsistent Timekeeping Due to Skipped Ethereum L1 Blocks
Asset	`agglayer-clock/src/block.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `BlockClock` implementation, which is designed to track time based on incoming blocks on Ethereum L1, sends out an `EpochEnded` event for every `epoch_duration` number of new blocks. This mechanism relies on the assumption that blocks are consistently generated at regular intervals.

However, because blocks can occasionally be skipped on Ethereum L1, the time intervals between block arrivals can vary. As a result, some epochs may take longer to complete than others. This inconsistency in block generation can lead to timing irregularities in protocols or applications relying on `BlockClock` for precise epoch measurements.

Such variability in epoch duration may impact any functionality that assumes predictable timing, potentially leading to performance issues or incorrect system behaviour.

Recommendations

Ensure the above comments are understood and consider changes if deemed applicable.

Resolution

The development team has acknowledged this finding with the following comment:

> *"This is understood and known, the goal of having block based clock is to rely on a decentralize pace on which multiple nodes can listen to.".*

AGG-22	`TimeClock` Starts From Epoch Zero
Asset	`agglayer-clock/src/time.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `Node` implementation utilizes `TimeClock::new_now()` to initialize a clock in the `start()` method, setting its genesis time to `Utc::now()`. Consequently, the current epoch is calculated as the time since genesis divided by the epoch duration. This means that the epoch count will reset to 0 each time the node is restarted.

While the `epoch` value is primarily used for logging purposes, this reset behaviour can lead to confusion and inconsistencies in log entries, as the epoch information may not accurately reflect the node's operational history.

Despite the testing team's classification of this as an informational issue, it is important to consider the potential for misinterpretation of logs in troubleshooting and monitoring scenarios.

Recommendations

Ensure the above comments are understood and consider changes if deemed applicable.

A possible solution would be setting a static genesis time in the config and using it to create a clock.

Resolution

The development team has acknowledged this finding with the following comment:

> *"The TimeClock is mainly used in tests or in local development test, the current default behaviour is to use the BlockClock."*

AGG-23	`SignedTx::hash()` does not contain `rollup_id`
Asset	`agglayer-node/src/signed_tx.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

`SignedTx::hash()` calculates the hash using the fields of the `SignedTx`. This includes values such as `last_verified_batch`, `new_state_root` and `proof`. However, it does not include `rollup_id`. As a result, two `SignedTx`s that have all fields equal to each other, except different `rollup_id` fields will have the same hash.

Crucially, the signature of a `SignedTx` is only based on the hash of that `SignedTx`. As a result, if two `SignedTx` have the same hash, a signature for one will also be valid for the other.

agglayer-node/src/signed_tx.rs

```rust
pub(crate) fn hash(&self) -> H256 {
  // ...

  let data = [
  last_verified_batch_hex.as_bytes(),
  new_verified_batch_hex.as_bytes(),
  &self.tx.zkp.new_state_root[..],
  &self.tx.zkp.new_local_exit_root[..],
  proof_hex.as_bytes(),
  ]
  .concat();

  keccak256(data).into()
}
```

`SignedTx`s are used to verify the Signer is the trusted sequencer of a rollup, but is also given as a format expected to be used in other, yet to be included systems.

Therefore, as it is not clear what the exact impact of this issue is, we rate this as an informational and recommend further investigation.

Recommendations

Consider adding more data, such as `rollup_id` to the hash calculation.

Resolution

The development team acknowledged this finding with the following comment:

> *"This is a backport from the first go version. It was implemented like that to have compatibility, and this will not be updated for this v0.1 FEP path."*

AGG-24	`CertificateOrchestrator::received_certificates` Will Grow Unbounded
Asset	`agglayer-certificate-orchestrator/src/lib.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

HashMap `received_certificates` in `CertificateOrchestrator` keeps all certificates received through `receive_certificate()`. As the certificates are never deleted from this map, it will keep growing for the entire runtime of the AggLayer process.

This could eventually lead to out-of-memory issues, which could then lead to the process terminating.

Given that this is very unlikely to be an issue in practical situations, we rate this finding as informational only.

Recommendations

Consider implementing an upper limit on the size of `received_certificates`. Old certificates could be deleted when they are no longer needed.

Resolution

The finding has been resolved in PR-267 by removing the `received_certificates` HashMap.

AGG-25	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Testing Private Keys Stored Unencrypted**

   Related Asset(s): agglayer-config/tests/backward_compatibility.rs, agglayer.toml

   Two key stores used for backwards compatiblity with old testing code are stored either unencrypted or with the password displayed in plaintext in the same config file.

   Do not store sensitive information, such as passwords or keys, in cleartext in configuration files. Instead, use their encrypted forms or hashes.

   Alternatively, opt in to read the value from the environment variables, however that puts certain trust assumptions on the operating environment.

2. **Unmaintained Crate Dependencies**

   Related Asset(s): agglayer-aggregator-notifier

   Rust advisory lists two crates that exist in the dependency list that are unmaintained:

   - ansi_term

   ```
   └── tracing-forest 0.1.6
   └──  sp1-core 1.1.1
   └──  sp1-sdk 1.1.1
   ├── pessimistic-proof-test-suite 0.1.0
   └── agglayer-aggregator-notifier 0.1.0
   └── agglayer-node 0.1.0
   └── agglayer 0.1.0
   ```

   - proc-macro-error

   ```
   proc-macro-error 1.0.4
   ├── alloy-sol-macro-expander 0.7.7
   └── alloy-sol-macro 0.7.7
   └── alloy-sol-types 0.7.7
   └── sp1-sdk 1.1.1
   ├── pessimistic-proof-test-suite 0.1.0
   └── agglayer-aggregator-notifier 0.1.0
   └── agglayer-node 0.1.0
   └── agglayer 0.1.0
   ```

   Unmaintained crates will not receive any updates. It is recommended to not use unmaintained crates for security and compatibility issues.

3. **Incorrect Comment**

   Related Asset(s): agglayer-node/src/kernel/mod.rs

   The comment on line [262] is misleading. `pendingStateNum` is a depreciated feature from the older version design, if populated with a non-zero value it would now simply cause the call to revert.

   It is suggested to update the comment to reflect that this feature is depreciated and not planned for future inclusion.

4. **Unchecked Panic With `unwrap()` If Fields Are Updated**

   Related Asset(s): agglayer/src/main.rs

   Use of `unwrap()` on the config on line [15] may become unsafe in the future if non-serializable fields are added.

   Ensure that no non-serializable fields are added to the config or remove the use of `unwrap()` in favour of pattern matching or custom error handling.

5. **Comparison To Boolean**

   Related Asset(s): PolygonRollupManager.sol

   There are comparisons to `true` on line [471], line [507] and line [720] of `PolygonRollupManager`.
   Consider checking the Boolean directly to reduce code complexity.

6. **Function Type Change**

   Related Asset(s): PolygonRollupManager.sol

   The function `getInputPessimisticBytes()` of `PolygonRollupManager` appears to be an external wrapper of an external function, so consider setting its visibility to `external`.

7. **Uncommented Parameter**

   Related Asset(s): PolygonRollupManager.sol

   The function `PolygonRollupManager.addNewRollupType()` has no NatSpec comment for its parameter `rollupVerifierType`.
   Consider adding a NatSpec description to make future development work easier.

8. **Typing Errors (Typos)**

   Related Asset(s): PolygonRollupManager.sol
   PolygonZkEVMGlobalExitRootV2.sol

   On line [43], line [76], line [148], line [414] and line [578] of `PolygonRollupManager.sol` the word "purpose" is spelled "porpuse".
   The comment on line [428] of `PolygonRollupManager.sol` should read, "No genesis on pessimistic rollups".
   The word "temporal" on line [81] of `PolygonZkEVMGlobalExitRootV2.sol` should be, "temporary".

9. **Large `NULLIFIER_TREE_DEPTH`**

   Related Asset(s): pessimistic-proof/src/nullifier_tree/mod.rs

   For performance reasons, consider setting `NULLIFIER_TREE_DEPTH` to a smaller value than currently configured 64, as per the corresponding `TODO` comment.

10. **Misleading Function Name And Behaviour - `verify_and_update()`**

    Related Asset(s): pessimistic-proof/src/local_balance_tree.rs, agglayer-node/src/signed_tx.rs

    The function `verify_and_update()` in the `pessimistic_proof` crate has a misleading name that suggests it performs an update operation. However, the function does not actually update any values, it only calculates a hypothetical root hash based on the assumption that values were updated. This could cause confusion for developers or users who expect the function to modify the tree.

    Likewise, the `SignedTx` struct in the `agglayer-node` crate is used for signature verification in general and is not always an Ethereum transaction, making the name misleading.

11. **Wrong Transaction Hash In Check**

    Related Asset(s): agglayer-node/src/kernel/mod.rs

    When attempting to settle a transaction on Ethereum L1, we check to see if the transaction hash as been seen before but the hash checked is the hash of the transaction proof not the hash of the transaction receipt.

    Currently this repeat transaction error logging is likely to never trigger.

    Ensure the correct transaction hash is checked when calling `check_tx_status()`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team has acknowledged the findings above and implemented some of the recommendations, as deemed appropriate, in the PR-341.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/PolygonPessimisticConsensus.t.sol:PolygonPessimisticConsensusTest
[PASS] test_getConsensusHash_vanilla() (gas: 16785)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 432.98ms (94.74µs CPU time)

Ran 8 tests for test/PolygonConsensusBase.t.sol:PolygonConsensusBaseTest
[PASS] test_acceptAdminRole_happy() (gas: 48262)
[PASS] test_acceptAdminRole_onlyPendingAdmin() (gas: 40910)
[PASS] test_setTrustedSequencerURL_happy() (gas: 29539)
[PASS] test_setTrustedSequencerURL_onlyAdmin() (gas: 16339)
[PASS] test_setTrustedSequencer_happy() (gas: 25219)
[PASS] test_setTrustedSequencer_onlyAdmin() (gas: 15679)
[PASS] test_transferAdminRole_happy() (gas: 42260)
[PASS] test_transferAdminRole_onlyAdmin() (gas: 15700)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 433.67ms (1.24ms CPU time)

Ran 6 tests for test/PolygonZkEVMGlobalExitRootV2.t.sol:PolygonZkEVMGlobalExitRootV2Test
[PASS] test_checkUpgrade() (gas: 78524)
[PASS] test_updateExitRoot_NotFromBridgeOrManager() (gas: 16210)
[PASS] test_updateExitRoot_rootAlreadyExists() (gas: 96028)
[PASS] test_updateExitRoot_updateMainnetRoot() (gas: 190394)
[PASS] test_updateExitRoot_updateRollupRoot() (gas: 190393)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 436.25ms (3.14ms CPU time)

Ran 43 tests for test/PolygonValidiumEtrog.t.sol:PolygonValidiumEtrogTest
[PASS] test_checkUpgrade() (gas: 58935)
[PASS] test_forceBatch_AboveMaxLength() (gas: 63523)
[PASS] test_forceBatch_NotDuringEmergency() (gas: 63814)
[PASS] test_forceBatch_NotEnoughPOL() (gas: 39138)
[PASS] test_forceBatch_OnlyAllowedSender() (gas: 33616)
[PASS] test_forceBatch_happy() (gas: 141988)
[PASS] test_sequenceBatchesValidium_exceedMaxBatches() (gas: 1572553)
[PASS] test_sequenceBatchesValidium_finalAccInputHashDoesntMatch() (gas: 158436)
[PASS] test_sequenceBatchesValidium_forcedBatchesMustMatch() (gas: 157093)
[PASS] test_sequenceBatchesValidium_happy() (gas: 244373)
[PASS] test_sequenceBatchesValidium_invalidL1InfoLeafIndex() (gas: 50320)
[PASS] test_sequenceBatchesValidium_invalidTimestamp() (gas: 25899)
[PASS] test_sequenceBatchesValidium_onlyTrustedSequencer() (gas: 22222)
[PASS] test_sequenceBatchesValidium_zeroBatches() (gas: 19584)
[PASS] test_sequenceBatches_exceedMaxBatches() (gas: 2517246)
[PASS] test_sequenceBatches_finalAccInputHashDoesntMatch() (gas: 153010)
[PASS] test_sequenceBatches_forcedDataMustMatch() (gas: 158812)
[PASS] test_sequenceBatches_happy() (gas: 242287)
[PASS] test_sequenceBatches_invalidL1InfoLeafIndex() (gas: 52648)
[PASS] test_sequenceBatches_invalidTimestamp() (gas: 28279)
[PASS] test_sequenceBatches_onlyTrustedSequencer() (gas: 19588)
[PASS] test_sequenceBatches_sequenceWithDataAvailabilityNotAllowed() (gas: 27232)
[PASS] test_sequenceBatches_transactionsTooLong() (gas: 1102290)
[PASS] test_sequenceBatches_zeroBatches() (gas: 19185)
[PASS] test_sequenceForceBatches_AfterEmergencyTimeout() (gas: 211223)
[PASS] test_sequenceForceBatches_BatchOverflow() (gas: 32369)
[PASS] test_sequenceForceBatches_ExceedMaxBatches() (gas: 2526938)
[PASS] test_sequenceForceBatches_ForceBatchTimeout() (gas: 134894)
[PASS] test_sequenceForceBatches_ForcedDataMismatch() (gas: 136185)
[PASS] test_sequenceForceBatches_OnlyAllowedSender() (gas: 179358)
[PASS] test_sequenceForceBatches_ZeroBatches() (gas: 29427)
[PASS] test_sequenceForceBatches_happy() (gas: 233531)
[PASS] test_setDataAvailabilityProtocol_onlyAdmin() (gas: 15701)
[PASS] test_setForceBatchAddress_forceBatchesDecentralized() (gas: 19680)
[PASS] test_setForceBatchAddress_happy() (gas: 25392)
[PASS] test_setForceBatchAddress_onlyAdmin() (gas: 15786)
[PASS] test_setForceBatchTimeout_happy() (gas: 38061)
```

```
[PASS] test_setForceBatchTimeout_higherThanAggregationTimeout() (gas: 16351)
[PASS] test_setForceBatchTimeout_higherThanPrevious() (gas: 81855)
[PASS] test_setForceBatchTimeout_onlyAdmin() (gas: 15802)
[PASS] test_switchSequenceWithDataAvailability_happy() (gas: 25954)
[PASS] test_switchSequenceWithDataAvailability_onlyAdmin() (gas: 19092)
[PASS] test_switchSequenceWithDataAvailability_sameValue() (gas: 19652)
Suite result: ok. 43 passed; 0 failed; 0 skipped; finished in 445.59ms (37.73ms CPU time)

Ran 30 tests for test/PolygonRollupManager.t.sol:PolygonRollupManagerTest
[PASS] test_PRMaddExistingRollup_chainAlreadyExists() (gas: 3025738)
[PASS] test_PRMaddExistingRollup_chainIdOutOfRange() (gas: 3025870)
[PASS] test_PRMaddExistingRollup_successPessimistic() (gas: 3202519)
[PASS] test_PRMaddExistingRollup_successStateTransition() (gas: 3152647)
[PASS] test_PRMaddNewRollupType_invalidPessimistic() (gas: 3099004)
[PASS] test_PRMaddNewRollupType_invalidStateTransition() (gas: 3098980)
[PASS] test_PRMaddNewRollupType_vanilla() (gas: 3186336)
[PASS] test_PRMcreateNewRollup_chainIdAlreadyExists() (gas: 3027182)
[PASS] test_PRMcreateNewRollup_chainOutOfRange() (gas: 3025075)
[PASS] test_PRMcreateNewRollup_doesNotExist() (gas: 3024495)
[PASS] test_PRMcreateNewRollup_typeObselete() (gas: 3035411)
[PASS] test_PRMcreateNewRollup_vanilla() (gas: 3851220)
[PASS] test_PRMrollbackBatches_invalidRollbackBatch() (gas: 3033963)
[PASS] test_PRMrollbackBatches_notAdmin() (gas: 3038093)
[PASS] test_PRMrollbackBatches_notEndOfSequence() (gas: 3067723)
[PASS] test_PRMrollbackBatches_notStateTransitionChain() (gas: 3591585)
[PASS] test_PRMrollbackBatches_rollupDoesntExist() (gas: 3118140)
[PASS] test_PRMrollbackBatches_success() (gas: 3071834)
[PASS] test_PRMupdateRollupByRollupAdmin_NotAdmin() (gas: 3100487)
[PASS] test_PRMupdateRollupByRollupAdmin_NotCompatible() (gas: 3171836)
[PASS] test_PRMupdateRollupByRollupAdmin_Obselete() (gas: 3118647)
[PASS] test_PRMupdateRollupByRollupAdmin_RollupDoesNotExist() (gas: 3187849)
[PASS] test_PRMupdateRollupByRollupAdmin_RollupTypeDoesNotExist() (gas: 3110422)
[PASS] test_PRMupdateRollupByRollupAdmin_Success() (gas: 3145917)
[PASS] test_PRMupdateRollupByRollupAdmin_TypeTooOld() (gas: 3101121)
[PASS] test_PRMupdateRollupByRollupAdmin_unverifiedSequences() (gas: 3024387)
[PASS] test_PRMverifyPessimisticTrustedAggregator_failingProof() (gas: 3679350)
[PASS] test_PRMverifyPessimisticTrustedAggregator_invalidGlobalExitRoot() (gas: 3661562)
[PASS] test_PRMverifyPessimisticTrustedAggregator_stateTransitionError() (gas: 3106228)
[PASS] test_PRMverifyPessimisticTrustedAggregator_success() (gas: 3863899)
Suite result: ok. 30 passed; 0 failed; 0 skipped; finished in 445.34ms (75.99ms CPU time)

Ran 30 tests for test/PolygonRollupBaseEtrog.t.sol:PolygonRollupBaseEtrogTest
[PASS] test_checkUpgrade() (gas: 58791)
[PASS] test_forceBatch_AboveMaxLength() (gas: 63457)
[PASS] test_forceBatch_NotDuringEmergency() (gas: 63770)
[PASS] test_forceBatch_NotEnoughPOL() (gas: 39050)
[PASS] test_forceBatch_OnlyAllowedSender() (gas: 33601)
[PASS] test_forceBatch_happy() (gas: 141861)
[PASS] test_sequenceBatches_exceedMaxBatches() (gas: 2515151)
[PASS] test_sequenceBatches_finalAccInputHashDoesntMatch() (gas: 150860)
[PASS] test_sequenceBatches_forcedDataMustMatch() (gas: 156542)
[PASS] test_sequenceBatches_happy() (gas: 239802)
[PASS] test_sequenceBatches_invalidL1InfoLeafIndex() (gas: 50486)
[PASS] test_sequenceBatches_invalidTimestamp() (gas: 26139)
[PASS] test_sequenceBatches_onlyTrustedSequencer() (gas: 17480)
[PASS] test_sequenceBatches_transactionsTooLong() (gas: 1100125)
[PASS] test_sequenceBatches_zeroBatches() (gas: 17134)
[PASS] test_sequenceForceBatches_AfterEmergencyTimeout() (gas: 210908)
[PASS] test_sequenceForceBatches_BatchOverflow() (gas: 32302)
[PASS] test_sequenceForceBatches_ExceedMaxBatches() (gas: 2526894)
[PASS] test_sequenceForceBatches_ForceBatchTimeout() (gas: 134674)
[PASS] test_sequenceForceBatches_ForcedDataMismatch() (gas: 135921)
[PASS] test_sequenceForceBatches_OnlyAllowedSender() (gas: 178978)
[PASS] test_sequenceForceBatches_ZeroBatches() (gas: 29405)
[PASS] test_sequenceForceBatches_happy() (gas: 233086)
[PASS] test_setForceBatchAddress_forceBatchesDecentralized() (gas: 19636)
[PASS] test_setForceBatchAddress_happy() (gas: 25326)
[PASS] test_setForceBatchAddress_onlyAdmin() (gas: 15764)
[PASS] test_setForceBatchTimeout_happy() (gas: 37922)
[PASS] test_setForceBatchTimeout_higherThanAggregationTimeout() (gas: 16262)
```

```
[PASS] test_setForceBatchTimeout_higherThanPrevious() (gas: 81687)
[PASS] test_setForceBatchTimeout_onlyAdmin() (gas: 15736)
Suite result: ok. 30 passed; 0 failed; 0 skipped; finished in 445.63ms (48.47ms CPU time)

Ran 8 test suites in 455.06ms (3.50s CPU time): 120 tests passed, 0 failed, 0 skipped (120 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

		Likelihood		
		Low	Medium	High
Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].