Polygon

AggOracleCommittee Contract

Security Assessment Report

Version: 2.0

August, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Polygon's `AggOracleCommittee` components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, 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 components in scope. 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*.

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

Overview

The `AggOracleCommittee` contract is responsible for managing the insertion of Global Exit Roots (GERs) into the `GlobalExitRootManagerL2SovereignChain`.

It operates as an oracle-based system where multiple oracle members vote on proposed GERs, and once a quorum is reached, the GER is consolidated into the `GlobalExitRootManagerL2SovereignChain`. Each oracle will have a single vote at a given time, meaning that an oracle cannot vote for 2 GERs simultaneously. Once a GER reaches the required number of votes (quorum), that GER will be consolidated.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Agglayer repository.

The scope of this time-boxed review was strictly limited to the following files at commit dc5bb8c:

- `contracts/v2/sovereignChains/AggOracleCommittee.sol`
- `deployment/v2/utils/updateVanillaGenesis.ts`
- `tools/createSovereignGenesis/create-sovereign-genesis.ts`
- `tools/deployAggOracleCommittee/deployAggOracleCommittee.ts`

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

Approach

The security assessment covered components written in Solidity.

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

Additionally, the manual review process 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.

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

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

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 3 issues during this assessment. Categorised by their severity:

- Medium: 2 issues.
- Low: 1 issue.

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 components, including 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 components(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
PAOC-01	Oracles Can Remove Other Oracles' Votes For A Consolidated GER	Medium	**Closed**
PAOC-02	Restriction On GER Re-Voting May Prevent Recovery From Incorrect Removals	Medium	**Closed**
PAOC-03	Missing Validation When Updating Quorum	Low	**Closed**

PAOC-01	Oracles Can Remove Other Oracles' Votes For A Consolidated GER		
Asset	`AggOracleCommittee.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Oracle votes can unintentionally overwrite other oracles' votes, undermining consensus integrity.

If an oracle re-votes for a previously consolidated Global Exit Root (`GER`), for example, when a valid `GER` is mistakenly removed from the `GlobalExitRootManagerL2SovereignChain` , the current vote tracking mechanism may cause other oracles' votes to be removed inadvertently or maliciously.

Consider the following scenario:

1. There are three oracle members: `[A,B,C]` , with a quorum of 2.
2. Oracle A votes for `GER1` , setting:
   - `addressToLastProposedGER[A] = GER1`
3. Oracle B also votes for `GER1` . This reaches the quorum, so:
   - `GER1` is consolidated
   - `addressToLastProposedGER[B] = INITIAL_PROPOSED_GER`

```
if (currentVotedReport.votes >= quorum) {
    _consolidateGlobalExitRoot(proposedGlobalExitRoot);
} else {
     // Store submitted report with a new added vote
     proposedGERToReport[proposedGlobalExitRoot] = currentVotedReport;

      // Store voted report hash
      addressToLastProposedGER[msg.sender] = proposedGlobalExitRoot;
}
```

4. The consolidated root is inserted into the `GlobalExitRootManagerL2SovereignChain` :

```
if (globalExitRootMap[_newRoot] == 0) {
    globalExitRootMap[_newRoot] = block.timestamp;
    //....
}
```

5. Later, suppose `GER1` is removed (either intentionally or by mistake) via a call to `GlobalExitRootManagerL2SovereignChain.removeGlobalExitRoots()` by `onlyGlobalExitRootRemover` :

```
delete globalExitRootMap[gerToRemove];
```

6. Realising the mistake, Oracle B votes again for `GER1` . Because B's previous vote was reset to `INITIAL_PROPOSED_GER` , it is allowed to vote for `GER1` again. Now:
   - `addressToLastProposedGER[B] = GER1`

- `proposedGERToReport[GER1].votes = 1`

7. Oracle A now votes for a different root, `GER2`. Since A previously voted for `GER1`, the following logic reduces `GER1`'s vote count:

```
// If it's not the initial report hash, check last report voted
if (lastProposedGER != INITIAL_PROPOSED_GER) {
    Report storage lastVotedReport = proposedGERToReport[
        lastProposedGER
    ];

    // Subtract a vote on the last voted report
    // That report could have 0 votes because:
    // - The report was already consolidated
    // - Were subtracted all the votes from that report
    if (lastVotedReport.votes > 0) {
        unchecked {
            lastVotedReport.votes--;
        }
    }
}
```

In scenario above, Oracle A's new vote unintentionally removes Oracle B's vote for `GER1`, despite B having voted independently. This undermines vote integrity, allowing one oracle's actions to affect another's vote history and count, potentially leading to incorrect consensus outcomes.

Recommendations

Update the vote tracking logic to prevent oracles from affecting the votes cast by the others.

Resolution

The finding has been closed with the following comment provided by the development team:

> *"The remove global exit root functionality is thought as a recovery/excepctional mechanism. So won't be an automatic process. Also the oracles will be coded to always vote for the last GER available in L1 which is the one that contains more information, and the other ones are redundant.*
>
> *In the scenario where we want to delete a GER, we probably stop the bridge, remove a GER and consider damages before restarting it. In all of this process a new GER will be available on L1 (either for a user deposit, or because any chain has verified a proof) and the oracles will vote for that GER.*
>
> *So, operationally this won't cause any additional problem, this scenario we hope that is extremely weird, and adding all the complexity to take in account removed GERs I think it's an overkill for this particular case.*
>
> *An important part of this is that, the new GER always contains all the information of the previous ones, meaning that, there's not much relevance if one GER was skipped to be inserted and the new ones are the ones that should be inserted since contain the new bridge information."*

PAOC-02	Restriction On GER Re-Voting May Prevent Recovery From Incorrect Removals		
Asset	`AggOracleCommittee.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In the current implementation, once an oracle votes for a given Global Exit Root (GER) and it gets consolidated, the same oracle is not allowed to vote for that GER again:

```
// Check if the proposed GER is not the same as the last voted report
require(
    lastProposedGER != proposedGlobalExitRoot,
    AlreadyVotedForThisGER()
);
```

This introduces an edge case where an oracle may be forced to vote for an invalid GER, or not vote at all.

Consider the following scenario:

1. Suppose there are three oracle members: `[A, B, C]`, with a quorum set to 2.
2. Oracle A votes for `GER1`. As a result:
   - `addressToLastProposedGER[A] = GER1`
   - `proposedGERToReport[GER1].votes = 1`
3. Oracle B then votes for `GER1`. Since the quorum is reached, `GER1` is consolidated. Now:
   - `addressToLastProposedGER[B] = INITIAL_PROPOSED_GER`
   - `proposedGERToReport[GER1].votes = 0`
4. On the `GlobalExitRootManagerL2SovereignChain`, the following entry is made:
   - `globalExitRootMap[GER1] = block.timestamp`

   ```
   if (globalExitRootMap[_newRoot] == 0) {
       globalExitRootMap[_newRoot] = block.timestamp;
       //...
   }
   ```
5. Later, `GER1` is removed, either mistakenly or maliciously, via `onlyGlobalExitRootRemover`, making:
   - `GlobalExitRootManagerL2SovereignChain.globalExitRootMap[GER1] = 0`

   ```
   // Remove the GER from the map
   delete globalExitRootMap[gerToRemove];
   ```
6. Oracle C then votes for a new root, `GER2`:

- `addressToLastProposedGER[C] = GER2`
- `proposedGERToReport[GER2].votes = 1`

7. Oracle B realizes that `GER1` was the correct root and attempts to vote for it again. Since its last vote was set to `INITIAL_PROPOSED_GER`, it is allowed to vote for `GER1`, updating:
   - `addressToLastProposedGER[B] = GER1`
   - `proposedGERToReport[GER1].votes = 1`
8. Oracle A also realizes `GER1` was the correct root and tries to vote for it again. However, because its `addressToLastProposedGER[A]` is already `GER1`, the system reverts:

```
require(
    lastProposedGER != proposedGlobalExitRoot,
    AlreadyVotedForThisGER()
);
```

Now, Oracle A is stuck with two bad options:

- **Option 1**: Vote for `GER2` (to reset its last voted GER) and then re-vote for `GER1`. This introduces two issues:
  - Since Oracle C already voted for `GER2`, voting for it again meets the quorum and consolidates an invalid root.
  - Voting for another GER resets the vote count for `GER1`, undoing Oracle B's vote.
- **Option 2**: Vote for an arbitrary new root like `GER3` just to reset state.
  - This too reduces `GER1`'s vote count to zero by removing Oracle B's vote, defeating the purpose of recovery.

Recommendations

Enhance the oracle voting logic by adding a mechanism that queries the state of `globalExitRootMap` from `GlobalExitRootManagerL2SovereignChain`.

This would allow an oracle to re-vote for a GER if it has been previously removed, thus supporting recovery from mistaken or malicious GER removals without corrupting the consensus process.

Resolution

The finding has been closed with the following comment provided by the development team:

> *"The remove global exit root functionality is thought as a recovery/excepctional mechanism. So won't be an automatic process. Also the oracles will be coded to always vote for the last GER available in L1 which is the one that contains more information, and the other ones are redundant.*
>
> *In the scenario where we want to delete a GER, we probably stop the bridge, remove a GER and consider damages before restarting it. In all of this process a new GER will be available on L1 (either for a user deposit, or because any chain has verified a proof) and the oracles will vote for that GER.*

So, operationally this won't cause any additional problem, this scenario we hope that is extremely weird, and adding all the complexity to take in account removed GERs I think it's an overkill for this particular case.

An important part of this is that, the new GER always contains all the information of the previous ones, meaning that, there's not much relevance if one GER was skipped to be inserted and the new ones are the ones that should be inserted since contain the new bridge information."

PAOC-03	Missing Validation When Updating Quorum		
Asset	`AggOracleCommittee.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

During contract initialisation, a check ensures the quorum does not exceed the number of oracle members:

```
require(
    _quorum <= _aggOracleMembers.length,
    QuorumCannotBeGreaterThanAggOracleMembers()
);
```

However, this constraint is missing in the `updateQuorum` function, allowing the owner to set a quorum higher than the current number of oracle members:

```
function updateQuorum(uint64 newQuorum) external onlyOwner {
    require(newQuorum != 0, QuorumCannotBeZero());

    quorum = newQuorum;
    emit UpdateQuorum(newQuorum);
}
```

This could lead to a scenario where achieving quorum is impossible, effectively blocking any action that depends on it.

Recommendations

Add a validation in the `updateQuorum` function to ensure the new quorum does not exceed the current number of oracle members.

Resolution

The finding has been closed with the following comment provided by the development team:

> *"Yes we don't check that since we want to give the maximum flexibility to the owner. And putting this check could make some operations more difficult.*
>
> *(Consider the scenario) I have a quorum 3/5 and I want not a 5/7. If I first add the oracle participants and then update the quorum, the contract will be unsecure until the quorum will be updated, since for a moment I will have a 3/7, and there are new, potentially malicious, oracle participants."*

Appendix A 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.

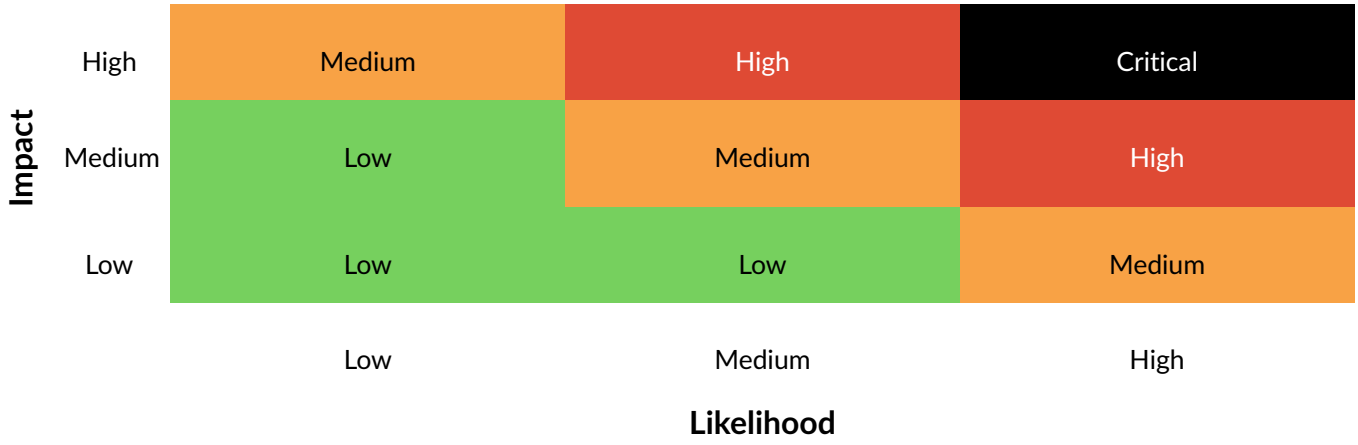

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].