Polygon

Agglayer Contracts v0.3.5

Security Assessment Report

Version: 2.0

September, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Polygon 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

AggLayer v0.3.5 introduces a new consensus type that addresses critical security limitations in previous versions by implementing multisig verification. It replaces the reliance on single ECDSA addresses for certificate submission with a hybrid solution that enables multiple signature verification whilst maintaining compatibility with existing infrastructure.

Core changes include updated bridge contracts, enhanced global exit root management, and new consensus infrastructure.

This release also introduces outpost chain support, allowing existing EVM-compatible chains like Base, Optimism, and BSC to integrate with the AggLayer whilst maintaining their own finality mechanisms.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to the smart contract changes between tag v11.0.0-rc.3 and tag v12.1.0-rc.3.

Additionally, the deployment script deployOutpostChain.ts was also reviewed.

The fixes of the identified issues were assessed at tag v12.1.0-rc.4.

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

Approach

The security assessment covered components written in Solidity.

For the Solidity components, 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 8 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.
- Low: 2 issues.
- Informational: 5 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 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
AG35-01	Inconsistent `gasTokenNetwork` Value In Deployment Output	Medium	**Resolved**
AG35-02	Oracle Removal Can Remove The Votes Of Other Oracles For Repeated Global Exit Root	Low	**Closed**
AG35-03	Signer Array Race Condition In Concurrent Updates	Low	**Closed**
AG35-04	Inaccurate `BridgeLib` Documentation Refers To Library	Informational	**Resolved**
AG35-05	Redundant `newFrontier` Parameter In `backwardLET` Function	Informational	**Closed**
AG35-06	Additional Getter Function Implementation Duplicates Public Variable	Informational	**Resolved**
AG35-07	Deployment Script Verification Error Blocks Analysis And Output	Informational	**Resolved**
AG35-08	Miscellaneous General Comments	Informational	**Resolved**

AG35-01	Inconsistent `gasTokenNetwork` Value In Deployment Output		
Asset	`deployOutpostChain.ts`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The deployment script generates inconsistent `gasTokenNetwork` values in the final output JSON, using `chainID` instead of `rollupID` despite all other script logic using `rollupID`.

Throughout the deployment script, `gasTokenNetwork` is consistently set to `deployParameters.network.rollupID`. The deployment logic explicitly assigns this value and includes a comment clarifying the relationship:

```
const gasTokenNetwork = deployParameters.network.rollupID; // Use rollupID as gasTokenNetwork
```

The bridge contract initialisation uses this value, and verification logic confirms the deployed contract matches this expected value:

```
if (Number(gasTokenNetwork) !== deployConfig.network.rollupID) {
    throw new Error(
        `Bridge gasTokenNetwork mismatch. Expected: ${deployConfig.network.rollupID}, Got: ${gasTokenNetwork}`,
    );
}
```

However, the final output generation function inconsistently uses `chainID` instead of `rollupID`:

```
return {
    deploymentDate: currentDateTime,
    network: {
        chainID: deployParams.network.chainID,
        rollupID: deployParams.network.rollupID,
        networkName: deployParams.network.networkName,
        gasTokenAddress: deriveGasTokenAddress(deployParams.network.rollupID),
        gasTokenNetwork: deployParams.network.chainID, // @audit Should be rollupID
    },
    // ... rest of output
};
```

This inconsistency creates a mismatch between the actual deployed contract configuration and the recorded deployment metadata. If the JSON output were relied on as the authoritative record for downstream operations such as proof generations or cross-chain bridge operations, the impact could potentially be significant.

Recommendations

Update the output generation function to use `deployParams.network.rollupID` instead of `deployParams.network.chainID` for the `gasTokenNetwork` field to maintain consistency with the deployed contract configuration and verification logic.

Resolution

Fixed by changing the output to use `rollupID` instead of `chainID` in commit 856c6e8.

AG35-02	Oracle Removal Can Remove The Votes Of Other Oracles For Repeated Global Exit Root		
Asset	`v2/sovereignChains/AggOracleCommittee.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Oracle member removal can inadvertently remove votes from different oracles in later voting rounds when the same Global Exit Root (GER) is voted on after consolidation.

When a GER is consolidated via `_consolidateGlobalExitRoot()`, the function deletes the report but leaves `addressToLastProposedGER` mappings unchanged for oracles who voted in that round:

```
function _consolidateGlobalExitRoot(bytes32 globalExitRoot) internal {
    // Delete the report
    delete proposedGERToReport[globalExitRoot]; // @audit Report deleted
    // Consolidate report
    globalExitRootManagerL2Sovereign.insertGlobalExitRoot(globalExitRoot);
    emit ConsolidatedGlobalExitRoot(globalExitRoot);
}
```

The `proposeGlobalExitRoot()` function intentionally avoids updating `addressToLastProposedGER` during consolidation (lines [**165-168**]), leaving stale mappings that point to consolidated GERs.

When the same GER is later voted on again, a new report entry is created. If an oracle from the original voting round is then removed, `removeOracleMember()` decrements votes from the new report based on the stale mapping:

```
// In removeOracleMember()
bytes32 lastVotedReportHash = addressToLastProposedGER[oracleMemberAddress]; // @audit Stale mapping
// ...
Report storage lastVotedReport = proposedGERToReport[lastVotedReportHash]; // @audit Points to new voting round
if (lastVotedReport.votes > 0) {
    lastVotedReport.votes--; // @audit Removes vote from different voting round
}
```

This creates cross-contamination where removing Oracle A (who voted in the earlier round) decrements the vote count for Oracle B (who voted in a later round on the same GER), effectively removing Oracle B's vote despite Oracle A having no participation in the latter voting round.

The likelihood is low as it requires the same GER to be voted on after consolidation, and the removed oracle must not have voted for any different GER in the interim period. Additionally, a different GER must be consolidated in between as the same GER cannot be proposed twice consecutively.

Recommendations

If there is no scenario where a previously consolidated GER would ever be legitimately proposed again then this issue may not need to be addressed. If such a scenario is possible, then clear `addressToLastProposedGER` mappings during consolidation or implement additional checks in `removeOracleMember()` to verify that the oracle actually participated in the current voting round before removing their vote.

Resolution

The development team acknowledged the issue and did not deem any modifications necessary at this time.

AG35-03	Signer Array Race Condition In Concurrent Updates		
Asset	`v2/AggLayerGateway.sol, v2/lib/AggchainBase.sol, v2/sovereignChains/AggOracleCommittee.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The signer removal mechanism creates potential race conditions when multiple transactions attempt to remove signers concurrently, despite existing protections within individual transactions.

Both `AggLayerGateway.sol` and `AggchainBase.sol` implement identical signer management logic through the `updateSignersAndThreshold()` function. The removal process uses a swap-and-pop pattern in `_removeSignerInternal()`:

```
function _removeSignerInternal(
    address _signer,
    uint256 _signerIndex
) internal {
    // ... validation code ...

    // Move the last element to the deleted spot and remove the last element
    aggchainSigners[_signerIndex] = aggchainSigners[signersLength - 1]; // @audit Array reordering

    aggchainSigners.pop();
}
```

Whilst the contracts implement descending order validation to prevent index conflicts within a single transaction, concurrent transactions can still interfere with each other:

- Transaction A is signed and submitted into the mempool. It will remove a signer at index 2.
- Transaction B is composed and signed. It will remove the signer at the last index.
- Transaction A is resolved, causing the last element to move to index 2.
- Transaction B now reverts as it became invalid due to the array reordering from Transaction A.

The existing mitigation requires indices in descending order and validates this requirement:

```
// Validate descending order of indices for removal to avoid index shifting issues
if (_signersToRemove.length > 1) {
    for (uint256 i = 0; i < _signersToRemove.length - 1; i++) {
        if (_signersToRemove[i].index <= _signersToRemove[i + 1].index) {
            revert IndicesNotInDescendingOrder();
        }
    }
}
```

However, this protection only applies within individual transactions, not across concurrent ones. The race condition leads to transaction failures with incorrect error messages, potentially confusing operators during critical multisig updates or causing automated systems to retry with stale indices.

There is a similar issue in `AggOracleCommittee.sol` which also implements oracle remover through a swap-and-pop pattern:

```
// Remove oracle member
addressToLastProposedGER[oracleMemberAddress] = bytes32(0);

// Remove the oracle member from the aggOracleMembers array
aggOracleMembers[oracleMemberIndex] = aggOracleMembers[
    aggOracleMembers.length - 1
];
aggOracleMembers.pop();

emit RemoveAggOracleMember(oracleMemberAddress);
```

Recommendations

Implement address-based removal instead of index-based removal to eliminate dependency on array positions. Alternatively, ensure that administrators are all aware of this potential race condition and that updates are performed with appropriate checks to ensure the race condition is never operative.

Resolution

The development team acknowledged the issue and did not deem any modifications necessary at this time.

AG35-04	Inaccurate `BridgeLib` Documentation Refers To Library
Asset	`v2/PolygonZkEVMBridgeV2.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Multiple comments in `PolygonZkEVMBridgeV2.sol` inaccurately refer to `BridgeLib` as a "library" when it is actually defined as a contract.

```
// Deploy the BridgeLib library
/// @dev this contract is used to store the bytecode of the BridgeLib library...
bridgeLib = new BridgeLib();
```

```
// Helpers to safely get the metadata from a token are now in BridgeLib library
```

`BridgeLib` is defined as `contract BridgeLib` rather than `library BridgeLib`. Whilst it functions as a library through external function calls, the documentation should accurately describe it as a contract for clarity.

Recommendations

Update comments to refer to `BridgeLib` as a "contract" rather than "library" to accurately reflect its implementation.

Resolution

Comments were updated to refer to `BridgeLib` as a contract rather than library in commit c74c68f.

AG35-05	Redundant `newFrontier` Parameter In `backwardLET` Function
Asset	`v2/sovereignChains/BridgeL2SovereignChain.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `newFrontier` parameter in the `backwardLET()` function is redundant as it can be entirely derived from the `proof` and `newDepositCount` parameters.

The function signature includes the `newFrontier` parameter:

```
function backwardLET(
    uint256 newDepositCount,
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata newFrontier, // @audit Redundant parameter
    bytes32 nextLeaf,
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata proof
) external onlyGlobalExitRootRemover ifEmergencyState
```

The parameter is used in two places: validation via `_checkValidSubtreeFrontier()` and tree rollback by setting `_branch[i] = newFrontier[i]`. However, the validation logic in `_checkValidSubtreeFrontier()` completely determines what `newFrontier` must contain based on `newDepositCount` and `proof`:

```
// From DepositContractBase.sol lines 168-182
while (index != 0 && height < _DEPOSIT_CONTRACT_TREE_DEPTH) {
    if ((index & 1) == 1) {
        // Must match proof sibling
        if (subTreeFrontier[height] != currentTreeProof[height]) {
            revert SubtreeFrontierMismatch();
        }
    } else {
        // Must be zero
        if (subTreeFrontier[height] != bytes32(0)) {
            revert NonZeroValueForUnusedFrontier();
        }
    }
    height++;
    index >>= 1;
}
```

Since `newFrontier` must exactly match `proof` at positions where bits are set in `newDepositCount` and be zero elsewhere, it provides no additional information beyond what can be computed from the other parameters.

Recommendations

Remove the `newFrontier` parameter and compute the frontier internally within the function based on `proof` and `newDepositCount`. This reduces interface complexity, eliminates potential caller errors, and may reduce gas costs.

Resolution

The development team acknowledged the issue and did not deem any modifications necessary at this time. They added the following comment to explain that the redundant parameter serves as a dual verification mechanism for security purposes:

```
/**
 * @dev Security Note: The `newFrontier` parameter is technically derivable from `newDepositCount` and `proof`,
 *      but is intentionally required as a dual verification mechanism. This forces callers to demonstrate
 *      complete understanding of the Merkle tree structure and acts as a safeguard against incorrect
 *      proof construction. The redundancy provides security-by-design for this critical emergency function.
 */
```

AG35-06	Additional Getter Function Implementation Duplicates Public Variable
Asset	`v2/AggLayerGateway.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract contains an explicit getter function that duplicates functionality already provided by Solidity's automatic getter generation for public variables.

Solidity automatically generates getter functions for public state variables. When a variable is declared as `public`, the compiler creates an external view function with the same name that returns the variable's value. The contract additionally implements a manual getter function that provides identical functionality.

```
/// @notice Threshold required for multisig operations
uint256 public threshold; // @audit Automatic threshold() getter created

function getThreshold() external view returns (uint256) {
    return threshold; // @audit Duplicates automatic getter
}
```

This redundancy increases contract size, deployment costs, and maintenance overhead whilst providing no functional benefit. External callers can use either the automatic or manual getters interchangeably, potentially causing confusion about the preferred interface.

Note that the function `getThreshold()` is currently used in `AggchainBase.sol`.

Recommendations

Consider declaring `threshold` as an `internal` variable as it is not required externally.

Resolution

The `threshold` variable was made `internal` to eliminate the redundant getter in commit dc924c4.

AG35-07	Deployment Script Verification Error Blocks Analysis And Output
Asset	`deployOutpostChain.ts`
Status	**Resolved:** See Resolution
Rating	Informational

Description

If there is any error in the verification steps of the deployment script, the script terminates, blocking all other checks and skipping output.

The `runBasicVerification()` function on lines [**648-669**] uses synchronous error propagation that terminates the entire script if any verification step fails. This occurs after all contract deployment transactions have been submitted and the script output is available in `outputJson`.

```
// Step 6: Run basic verification
logger.info('\n=== Step 6: Running verification ===');
await runBasicVerification(deployParameters, outputJson); // @audit Script exits here if verification fails

// Step 7: Generate final output
logger.info('\n=== Step 7: Generating deployment output ===');
const finalOutput = generateFinalOutput(outputJson, deployParameters, globalExitRootUpdater);
// @audit This code never executes if verification fails
```

The verification function throws errors directly without error handling:

```
async function runBasicVerification(deployConfig: any, outputJson: any) {
    // Step 1: Verify ProxyAdmin Contract
    await verifyProxyAdminContract(deployConfig, outputJson); // @audit Throws on failure

    // Step 2: Verify Timelock Contract
    await verifyTimelockContract(deployConfig, outputJson); // @audit Throws on failure
    // ... additional verification steps
}
```

When verification fails, users lose access to deployment information such as contract addresses, deployment configuration, and structured output needed for future operations. Additionally, an error is of course indicative of there being a problem with deployment, and so it would be beneficial to run the additional verification checks and output a record of their results.

Recommendations

Implement individual error handling within the verification function to ensure deployment output generation always succeeds. Wrap each verification step in `try-catch` blocks and return a summary of verification results rather than throwing errors. This approach maintains script continuity whilst providing detailed feedback on verification status.

Resolution

The recommendation was implemented: the deployment script was extensively modified to handle verification errors gracefully while still generating output in commit d504ceb.

AG35-08	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **Typo**

   Related Asset(s): v2/PolygonZkEVMBridgeV2.sol

   On line [**932**] contains a grammatical error in the comment: "check that this global exit root exist". It should say "exists" for proper English grammar.

   Update the comment to "check that this global exit root exists".

2. **Obsolete Commented Code References Undefined Variable**

   Related Asset(s): v2/lib/BridgeLib.sol

   The `validateAndProcessPermit()` function contains commented out code that references an undefined `amount` variable.

   ```
   // if (value != amount) {
   //     revert NotValidAmount();
   // }
   ```

   The preceding comment indicates this check was intentionally removed for OpenZeppelin's ERC20 compatibility, but the commented code remains and references a non-existent variable. This makes the comment a source of confusion rather than clarification.

   Remove the obsolete commented code.

Recommendations

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

Resolution

The development team's responses to the raised issues above are as follows:

1. **Typo** - Fixed by correcting the grammar from "exist" to "exists" in commit 76b2b5d.
2. **Obsolete Commented Code References Undefined Variable** - Fixed by removing the obsolete commented code that referenced the undefined `amount` variable in commit 76b2b5d.

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.

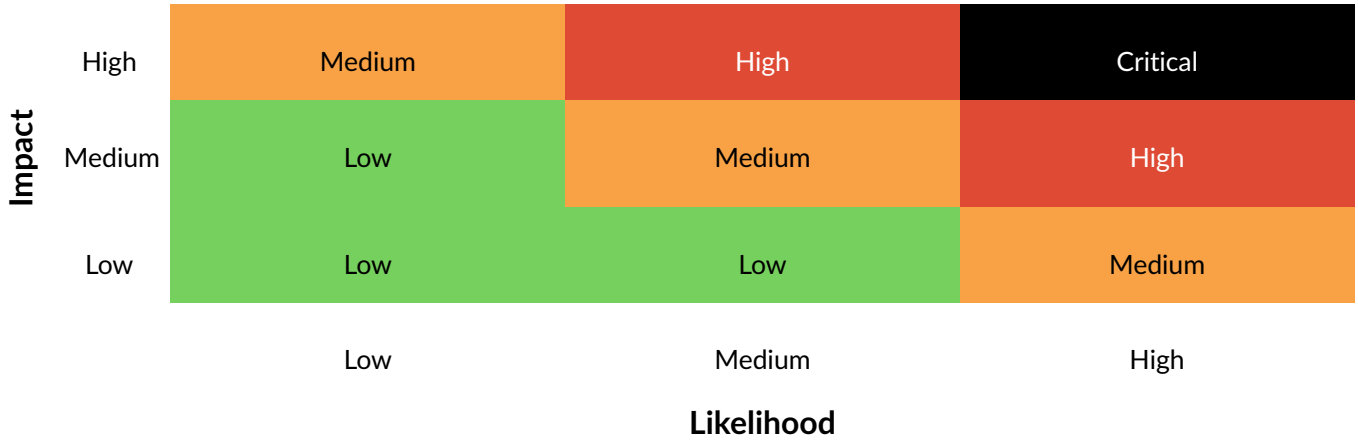

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].