Polygon

AggLayer v0.3.0 - Smart Contract Updates

Security Assessment Report

Version: 2.1

April, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of selected Polygon components. The review focused solely on the security aspects of the Solidity implementation of the contracts, 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*.

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.

The AggLayer v0.3.0 update specifically introduces support for pessimistic proofs. To achieve this, several contracts have been modified and new contracts have been created to accommodate pessimistic proof validation. This version allows any chain to define its own specific requirements, as long as it follows the generic interface outlined by AggLayer. As a result, the AggLayer becomes agnostic to specific chain requirements such as whether they are EVM-based, non-EVM or follow other specific consensus mechanisms.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the agglayer agg-contracts-internal repository.

The scope of this time-boxed review was strictly limited to `agg-contracts-internal` repository tagged v10.0.0-rc.2 and specifically the following contracts:

- `AggchainECDSA.sol`
- `AggchainFEP.sol`
- `AggLayerGateway.sol`
- `PolygonRollupManager.sol` changes from `v9.0.0-rc.5-pp`
- `BridgeL2SovereignChain.sol` changes from `v9.0.0-rc.5-pp`
- `GlobalExitRootManagerL2SovereignChain.sol` changes from `v9.0.0-rc.5-pp`

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

Retesting

Retesting was conducted on each identified finding using the revlevant commit or pull request. For details on the specific commit references and changes, please refer to the *Resolution* section of each finding.

The retesting scope was also further extended to include the following items:

- updateVanillaGenesis.ts @ 64d75de
- PR-5
- PR-7
- PR-8
- PR-12
- PR-13
- PR-14
- PR-15
- PR-17
- PR-18
- PR-25
- PR-26
- PR-32
- PR-36
- PR-38
- PR-39

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 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`

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 27 issues during this assessment. Categorised by their severity:

- Medium: 4 issues.
- Low: 7 issues.
- Informational: 16 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 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
AGL3-01	Missing Support For Non-Standard ERC20 Tokens With Custom Transfer Behaviour	Medium	Resolved
AGL3-02	Upgrades To Old `ALGateway` Type Using `updateRollupByRollupAdmin()` Will Revert	Medium	Resolved
AGL3-03	Potential Denial-Of-Service (DoS) Due To Hitting `LeafCount` Maximum Value	Medium	Closed
AGL3-04	Inability To Upgrade Valid Rollup Types	Medium	Resolved
AGL3-05	Blockchains With Addresses Bigger Than 20 Bytes Will Not Be Supported	Low	Closed
AGL3-06	Fixed-point Arithmetic Required For `l2BlockTime`	Low	Closed
AGL3-07	No Check for Self-Reference in `consensusImplementation` Assignment	Low	Resolved
AGL3-08	Missing Contract Verification for `rollupAddress` and `verifier`	Low	Resolved
AGL3-09	Missing `nonReentrant` Modifier on `rollbackBatches()`	Low	Resolved
AGL3-10	Missing Rollup Existence Checks Across Manager Functions	Low	Closed
AGL3-11	`AggLayerGateway` Proxy Is Inheriting A Non-Upgradeable Contract	Low	Closed
AGL3-12	Missing Monotonicity Checks for Pessimistic Chains (Non-ALGateway Types)	Informational	Closed
AGL3-13	ERC20s That Upgrade Their Decimals Would Break Accounting	Informational	Closed
AGL3-14	Migration Of Native Tokens Is Not Supported	Informational	Resolved
AGL3-15	Lack Of Zero Address Checks	Informational	Resolved
AGL3-16	Single-Step Role Transfers	Informational	Resolved
AGL3-17	Lack Of `aggChainData` Length Check	Informational	Resolved
AGL3-18	Lack Of `proofBytes` Length Check	Informational	Resolved
AGL3-19	Events Emitted Prior to Successful `initialize()` Call	Informational	Resolved
AGL3-20	Inconsistencies With Error Name	Informational	Resolved
AGL3-21	Unnecessary Check At `PolygonRollupManager`	Informational	Closed
AGL3-22	Consensus Implementations Storage Layout Could Be Namespaced To Increase Upgrade Safety	Informational	Resolved
AGL3-23	Missing Event Emission At `GlobalExitRootManagerL2SovereignChain` Initialization	Informational	Resolved

AGL3-24	Missing Gap On `GlobalExitRootManagerL2SovereignChain`	Informational	**Resolved**
AGL3-25	Common Infinite Approval Via `permit()` Is Incompatible With `bridgeAsset()` Calls	Informational	**Resolved**
AGL3-26	Missing Permit Bridging Functionality For Non-native Tokens	Informational	**Resolved**
AGL3-27	Miscellaneous General Comments	Informational	**Resolved**

AGL3-01	Missing Support For Non-Standard ERC20 Tokens With Custom Transfer Behaviour		
Asset	`contracts/v2/sovereignChains/BridgeL2SovereignChain.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `BridgeL2SovereignChain.sol` contract does not properly account for ERC20 tokens that have non-standard transfer behaviors, leading to a potential loss of funds.

As implemented in `PolygonZkEVMBridgeV2.sol#L271`, certain ERC20 tokens require additional handling when transferred. This is necessary for:

- **Fee-on-transfer** tokens (more info here).
- **Max-value amount transfers user balance** tokens (more info here).

Since these tokens do not always transfer the full amount specified in `amount`, their balance before and after transfer must be checked to determine the actual amount received.

However, this check is missing in `BridgeL2SovereignChain.sol`, which could lead to funds being lost or stolen.

The following functions have been identified to be affected:

1. `_bridgeWrappedAsset()` - missing check at `BridgeL2SovereignChain.sol` on line [**414**].
2. `_claimWrappedAsset()` - missing check at `BridgeL2SovereignChain.sol` on line [**440**].

This could lead to the following, depending on the type of token:

1. **Fee-on-transfer tokens**
   - If a token deducts a fee during transfer, the amount received will be less than the amount specified.
   - Without the correct balance check, the wrong value will be stored in the bridge, causing accounting errors.
2. **Max-value transfer tokens**
   - Some ERC20 tokens send the entire balance when `uint256.max` is passed as the amount.
   - If the bridge records this `uint256.max` in its Merkle leaf, it will result in all funds being drained when finishing bridging on destination.

This will not happen with all tokens as the logic of the affected functions only triggers when:

- The token being bridged is originally from another network.

- The token was updated to a new sovereign address (`_setSovereignTokenAddress()`) to ensure that its mechanics, such as transfer fees, remain the same as in the original network.

Note: Currently, all bridged tokens are deployed as `TokenWrapped` contracts, which do not retain special mechanics. However, the bridge permits updating token addresses, allowing reintroduction of non-standard behaviour.

Recommendations

Implement *"balance-before"* and *"balance-after"* accounting in the affected functions:

1. Check the contract's balance before the token transfer.
2. Check the balance after the transfer.
3. Compute the actual amount received by taking the difference.
4. Store this correct value as the `leafAmount` in the Merkle tree.

This approach ensures accurate accounting and prevents fund loss due to non-standard ERC20 behaviour.

Resolution

The suggested changes were made in 07f7136.

AGL3-02	Upgrades To Old `ALGateway` Type Using `updateRollupByRollupAdmin()` Will Revert		
Asset	`/agglayer/agg-contracts-internal/contractsPolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

One upgrade condition from rollups to `ALGateway` is incorrectly implemented, causing certain valid upgrades to revert unexpectedly.

The issue lies in the condition at `PolygonRollupManager.sol` line [**794**].

The current logic prevents upgrades to an older `newRollupTypeID` even when the destination is actually an `ALGateway`, which should be allowed as per the comments on the code.

```
// Only allowed to update to an older rollup type id if the destination rollup type is ALGateway
if (rollup.rollupTypeID >= newRollupTypeID) {
    revert UpdateToOldRollupTypeID();
}
```

While this check correctly blocks updates to older `rollupTypeIDs`, it misses the necessary exception that allows the downgrade if the new rollup type is `ALGateway`.

As a result, rollup admins are unable to upgrade to an older `ALGateway` as intended via the `updateRollupByRollupAdmin()` function.

> Note: This functionality isn't entirely lost, as roles with `_UPDATE_ROLLUP_ROLE` can still perform the update through `updateRollup()`.

Recommendations

The conditional check should be updated to the following:

```
        // Only allowed to update to an older rollup type id if the destination rollup type is ALGateway
        if (rollup.rollupTypeID >= newRollupTypeID
+           && rollupTypeMap[newRollupTypeID].rollupVerifierType != VerifierType.ALGateway) {
            revert UpdateToOldRollupTypeID();
        }
```

Resolution

In 2938b39, comments were changed in the code to indicate that no updates to older rollup types should be possible in this function.

AGL3-03	Potential Denial-Of-Service (DoS) Due To Hitting `LeafCount` Maximum Value		
Asset	`BridgeL2SovereignChain.sol, PolygonZkEVMBridgeV2.sol, lib/DepositContractBase.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The leaf index in `DepositContractBase._addLeaf()` on line [**69**] is restricted by `uint32`, with the maximum number of deposits it can track being `MAX_DEPOSIT_COUNT = 2**32 - 1` (4.2 billion):

```
if (depositCount >= _MAX_DEPOSIT_COUNT) {
    revert MerkleTreeFull();
}
```

Hitting this limit will result in reverts on all bridge functions, as each of them call `_addLeaf()`, rendering the contracts unusable.

It is possible this limit will be reached, based on the following calculations:

- Min gas for `bridgeAsset()` tx: `50,001 gas` (as per `gas_report.md`)
- Polygon zkEVM block gas limit: `30,000,000`
- Max transactions per block: `30,000,000 / 50,001 = 599 tx/block`
- Assuming using 10% of the block: `59 tx/block`
- Leaves added per block: `59`
- Blocks required to exhaust the index `(2**32 - 1) / (59 leaves/block) = 72,796,055 blocks`

At 3.46s per block, this limit will be reached in approximately **8 years** at 10% block usage (with a higher block usage or block gas limit, this could occur much faster).

Since this contract will be deployed on multiple L2s, each with different gas limits and TPS, this problem could arise quicker.

Recommendations

A solution is to increase the `MAX_DEPOSIT_COUNT`, using the maximum `uint64` would allow for over 10^{19} leaves.

Alternatively, consider implementing a circular buffer allowing the earliest leaves to be overwritten after the max amount is reached. However, this is not ideal either as it would result in some data loss - as this would happen once every few years, consider if this could be a sufficient time for the first users to claim their deposits before being removed.

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-04	Inability To Upgrade Valid Rollup Types		
Asset	`contracts/v2/PolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`PolygonRollupManager` has two types of IDs: `RollupID` and `RollupTypeID`. Each has its respective mappings and represents different concepts:

- `RollupID` is the ID of the rollup chain, effectively acting as a nonce that tracks the number of rollups added.
- `RollupTypeID` represents the type of rollup, tracking the different categories of rollups added.

However, on line [730], the function `updateRollupByRollupAdmin()` performs the following operation:

```
function updateRollupByRollupAdmin(
    ITransparentUpgradeableProxy rollupContract,
    uint32 newRollupTypeID
) external {
    // code...

    uint32 rollupID = rollupAddressToID[address(rollupContract)];
    // Admin can't update to different rollup type
    if (
         // @audit, using a rollupID on the rollupType mapping
        rollupTypeMap[rollupID].rollupVerifierType !=
        rollupTypeMap[newRollupTypeID].rollupVerifierType
    ) {
        revert UpdateNotCompatible();
    }

    // code...
}
```

The issue arises because arbitrary data, detached from the `rollupID`, is being accessed incorrectly. The data being read actually belongs to a `rollupTypeID`, leading to an inconsistency. This could result in an admin being unable to update a rollup to a valid different type.

By examining the current `PolygonManagerContract` on-chain state, consider a scenario of updating `rollupID = 10` to `rollupTypeID = 8`, which should be possible because:

- Reading `RollupID == 10` from `rollupIDToRollupDataV2()` returns `0` as the `rollupVerifierType` (third argument returned from the end). Therefore, it should be updatable to any `rollupTypeID` with `rollupVerifierType == 0`.
- Reading `RollupTypeID == 8` from `rollupTypeMap()` returns `0` as the `rollupVerifierType`. So `rollupID 10` should be updatable to `rollupTypeID 8`.
- Reading `RollupTypeID == 10` from `rollupTypeMap()` returns `1` as the `rollupVerifierType`.

However, the current code will execute as follows:

```
> 1st    rollupTypeMap[rollupID].rollupVerifierType != rollupTypeMap[newRollupTypeID].rollupVerifierType
> 2nd    rollupTypeMap[10].rollupVerifierType != rollupTypeMap[8].rollupVerifierType
> 3rd    1 != 0
```

Recommendations

Use `_rollupIDToRollupData` mapping instead to correctly reference the rollup's verifier type, e.g.:

```
_rollupIDToRollupData[rollupID].rollupVerifierType
```

Resolution

The development has fixed this issue in commit a90f60b by changing the condition to:

```
// Admin can't update to different rollup type
  if (
    rollup.rollupVerifierType !=
    rollupTypeMap[newRollupTypeID].rollupVerifierType
) {
    revert UpdateNotCompatible();
}
```

AGL3-05	Blockchains With Addresses Bigger Than 20 Bytes Will Not Be Supported		
Asset	`BridgeL2SovereignChain.sol, PolygonZkEVMBridgeV2.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Bridging functions that handle token transfers and messages currently use Solidity's `address` data structure, which is limited to 20 bytes.

However, some blockchain networks, such as Solana, require larger address formats (e.g. 32 bytes). While Solana itself is not ZK-based, AggLayer aims to support any chain, including potential ZK-enabled chains with larger addresses.

If a chain with addresses longer than 20 bytes is introduced, the bridge will not support it because it cannot properly store or process those addresses.

Recommendations

Use `bytes` instead of `address` in bridging functions. Modify all bridging functions to replace `address` with `bytes`, ensuring compatibility with variable-length addresses.

```
- address destinationAddress;
+ bytes destinationAddress;
```

Also, consider the following:

1. **Encoding Issues**
   - Be cautious when using `abi.encodePacked()` with `bytes` data structures.
   - Ensure encoding functions handle variable-length addresses correctly.
2. **Address Size Detection**
   - Implement logic to detect whether an address is **20 bytes (Ethereum-style)**.
   - If **20 bytes**, cast it to `address` for simplicity.
   - If longer, store and handle it as a `bytes` object of the required size.

This approach future-proofs the bridge, allowing it to support a wider range of blockchain ecosystems.

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-06	Fixed-point Arithmetic Required For `l2BlockTime`		
Asset	`contracts/v2/aggchains/AggchainFEP.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `l2BlockTime` variable in `AggchainFEP` is insufficient for some chains.

Currently, `l2BlockTime` has a minimum representation of 1 second, which is problematic for chains with sub-second block times. This can be inferred from `AggchainFEP.sol` on line [**534**], where time is used in calculations without fixed-point arithmetic to handle decimal values.

Since L2 chains prioritize speed, sub-second block times should be expected. If a chain with a block time of less than 1 second is created, the `getAggchainHash()` function would compute a hash with a higher than intended value. This hash is crucial for ZK proof verification, as seen in `PolygonRollupManager.sol` on line [**1515**], where it is read and then returned at `PolygonRollupManager.sol` on line [**1203**]. The returned value is subsequently used for verification in an `if` statement, potentially leading to invalid proofs or unexpected behavior.

Additionally, some chains use fractional values for their block times (e.g., 1.5s). The current representation lacks precision and will introduce inaccuracies even for chains with block times greater than 1 second.

Recommendations

Use fixed-point arithmetic, similar to ERC20 token decimals, to represent `l2BlockTime`. This allows for precise representation of block times, including fractional values.

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-07	No Check for Self-Reference in `consensusImplementation` Assignment		
Asset	`contracts/v2/PolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `attachAggchainToAL()` function does not validate that the `consensusImplementation` address is not set to `address(this)`.

Assigning the implementation to the proxy's own address would create a self-referential loop, potentially causing infinite `delegatecalls` or unexpected behaviour during upgrades.

Recommendations

Implement validation to ensure that `consensusImplementation` is not equal to `address(this)` before assignment, e.g.:

```
if (consensusImplementation == address(this)) revert InvalidImplementationAddress();
```

Resolution

The suggested changes were made in 3643c58.

AGL3-08	Missing Contract Verification for `rollupAddress` and `verifier`		
Asset	`contracts/v2/PolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `addExistingRollup()` function does not validate whether `rollupAddress` and `verifier` point to deployed contracts.

Without such checks, these parameters could be set to externally owned accounts (EOAs) or invalid addresses, leading to misconfiguration or runtime errors when attempting contract interactions.

Recommendations

Add checks to verify that both `rollupAddress` and `verifier` contain deployed contract code, e.g.:

```
if (rollupAddress.code.length == 0) revert InvalidRollupAddress();
if (verifier.code.length == 0) revert InvalidVerifierAddress();
```

Resolution

The suggested changes were made in eb96842.

AGL3-09	Missing `nonReentrant` Modifier on `rollbackBatches()`		
Asset	`contracts/v2/PolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `rollbackBatches()` function on line [**892**] performs an external call to `rollupContract.rollbackBatches()`.

Without a `nonReentrant` modifier or reentrancy guard, it is technically possible for a malicious rollup contract to re-enter the manager contract during this call, particularly in scenarios involving an upgrade to a compromised rollup implementation.

Although no immediate reentrancy vector is evident in the current logic, the absence of a reentrancy guard introduces unnecessary risk in the event when a rollup contract is upgraded to a malicious implementation.

Recommendations

Apply the `nonReentrant` modifier from OpenZeppelin's `ReentrancyGuard`, or equivalent protection, to the `rollbackBatches()` function to ensure the function cannot be re-entered during execution, mitigating the risk of future reentrancy.

Resolution

The suggested changes were made in f9825c6.

AGL3-10	Missing Rollup Existence Checks Across Manager Functions		
Asset	`contracts/v2/PolygonRollupManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Multiple functions in `PolygonRollupManager` assume the existence of a rollup entry without verifying whether it has been properly initialised or registered. Specifically:

- In `verifyPessimisticTrustedAggregator()`, there is no check to ensure that `rollup.rollupContract != address(0)`. This allows function execution to proceed with a default-initialised rollup, potentially resulting in unintended interactions or state updates involving the zero address.
- In view functions such as `getLastVerifiedBatch()`, `getRollupBatchNumToStateRoot()` and `getRollupSequencedBatches()`, absence of rollup existence checks may lead to misleading return values (e.g. returning 0 as the last verified batch when no rollup was ever registered).
- The function `rollupIDToRollupDataDeserialized()` also lacks validation to ensure that the rollup ID maps to a valid entry.

Recommendations

Add explicit rollup existence checks at the beginning of all relevant functions to prevent misuse or reliance on uninitialised state, e.g.:

```
if (rollup.rollupContract == address(0)) revert RollupDoesNotExist();
```

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-11	`AggLayerGateway` Proxy Is Inheriting A Non-Upgradeable Contract		
Asset	`AggLayerGateway.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `AggLayerGateway` contract functions as a proxy but inherits from a non-upgradeable contract, `AccessControl`.

The `AccessControl` contract lacks storage gaps, which introduces a risk of storage layout corruption during future upgrades.

Recommendations

Replace the current inheritance with `AccessControlUpgradeable` from OpenZeppelin Contracts Upgradeable to ensure safer long term upgradeability.

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-12	Missing Monotonicity Checks for Pessimistic Chains (Non-ALGateway Types)
Asset	`contracts/v2/PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

For rollup types using `VerifierType.Pessimistic`, there is currently no onchain enforcement that submitted values such as `newPessimisticRoot` or `newLocalExitRoot` progress monotonically over time (e.g., increasing L2 block number, timestamp, or Merkle root chain). In contrast, `VerifierType.ALGateway` implementations such as `AggchainFEP` do enforce ordering constraints using L2 block metadata.

For non-ALGateway pessimistic chains, the `PolygonRollupManager` delegates verification to the rollup's configured circuit via `ISP1Verifier.verifyProof()`. However, monotonicity is only guaranteed if the underlying SP1 circuit explicitly proves it as this is not enforced or asserted at the contract level. The default consensus logic (`getConsensusHash()`) simply returns a static configuration hash and provides no guarantees about monotonic progression.

Note, underlying risks are partially mitigated by access control as only entities with `_TRUSTED_AGGREGATOR_ROLE` may call `verifyPessimisticTrustedAggregator()`.

Recommendations

Enforce the following:

1. **Prove monotonicity in the SP1 circuit**
   - Ensure that `newPessimisticRoot` and `newLocalExitRoot` are derived from the prior state and provably ordered. This includes encoding the previous pessimistic root, timestamp, or batch ID as part of the witness and enforcing increasing order.
2. **Bind rollup verifier to strict circuit versions**
   - If multiple circuits are supported, contracts should pin `programVKey` to circuit versions that are known to enforce monotonicity.

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-13	ERC20s That Upgrade Their Decimals Would Break Accounting
Asset	`contracts/v2/sovereignChains/BridgeL2SovereignChain.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

If a token upgrades its decimals on the origin chain, the accounting for that token would break across all AggLayer chains.

While rare, this scenario is possible for upgradeable tokens. If a token's decimals increase or decrease during an upgrade, and a migration is subsequently required on other chains, incorrect amounts will be minted and burned due to the discrepancy.

Currently, the migration method allows name and symbol updates, but decimals remain unaccounted for in migrations. This means that even if a token is successfully updated, its accounting remains broken, leading to potential fund miscalculations.

Recommendations

Validate decimals on migration. Modify the migration function to verify that decimals remain unchanged. If decimals do change, automatically adjust mint and burn migration amounts based on the new decimal format.

This check should occur before the execution of the mints and burns, starting at `BridgeL2SovereignChain.sol` line [**361**].

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-14	Migration Of Native Tokens Is Not Supported
Asset	`contracts/v2/sovereignChains/BridgeL2SovereignChain.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The documentation for `migrateLegacyToken()` states that it allows migration of old native tokens, as seen in the comment:

```
@notice Moves old native or remapped token (legacy) to the new mapped token.
```

However, the current code does not support this due to the following check:

```
if (legacyTokenInfo.originTokenAddress == address(0)) {
    revert TokenNotMapped();
}
```

This prevents migration of native tokens since their `originTokenAddress == address(0)`, making the documentation misleading or the implementation wrong.

Recommendations

If migrating native tokens is intended, modify the function logic to support it.

If not intended, update the documentation to clarify that only remapped (legacy) tokens can be migrated.

Resolution

The code comments were modified to clarify that only remapped (legacy) tokens can be migrated. The changes were made in 443b493.

AGL3-15	Lack Of Zero Address Checks
Asset	`contracts/v2/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Several functions across the codebase do not validate whether an input address is the zero address (`address(0)`).

The following functions are affected (note, the list below should not be considered exhaustive):

- In `aggchains/AggchainECDSA.sol` , the `constructor()` function.
- In `aggchains/AggchainECDSA.sol` , the `initialize()` function, particularly for parameters affecting critical roles, such as `trustedSequencer` , `vKeyManager` , `admin` .
- In `aggchains/AggchainFEP.sol` , the `constructor()` function.
- In `aggchains/AggchainFEP.sol` , the `_initializeAggchain()` function does not verify addresses in `_initParams` parameter, particularly ones for critical roles, such as `aggChainManager` or `optimisticModeManager` .
- In `aggchains/AggchainFEP.sol` , the `transferAggChainManagerRole()` and `transferOptimisticModeManagerRole()` functions.
- In `AggLayerGateway.sol` , the `initialize()` function.
- In `AggLayerGateway.sol` , the `addPessimisticVKeyRoute()` function and `verifier` parameter. This may have larger implications as the surrounding logic suggests that the `route.verifier` should not be set, implying only update is expected. With `verifier` remaining as a zero address, it would make it appear as if the route still does not exists, affecting downstream logic (e.g. such as that in `freezePessimisticVKeyRoute()`).
- In `AggLayerGateway.sol` , the `addDefaultAggchainVKey()` , `freezePessimisticVKeyRoute()` and `updateDefaultAggchainVKey()` functions. In mappings, a key of `bytes4(0)` is valid, but it is often used to assume an unset value. Currently someone could unintentionally register keys under selector `0x00000000` .
- In `sovereignChains/BridgeL2SovereignChain.sol` , the `setBridgeManager()` function does not validate `_bridgeManager` parameter, which could render this role unusable. Could also consider implementing 2-step process of transferring the ownership.
- In `sovereignChains/BridgeL2SovereignChain.sol` , the `_claimWrappedAsset()` function may burn assets if `destinationAddress` is a zero address.
- In `sovereignChains/GlobalExitRootManagerL2SovereignChain.sol` , the `initialize()` functions does not validate `globalExitRootRemover` . If `globalExitRootRemover` is accidentally set to `address(0)` , it cannot be changed afterwards, as the function `setGlobalExitRootRemover()` has the `onlyGlobalExitRootRemover` modifier, preventing the function `removeGlobalExitRoots()` from being called.

Failing to check for the zero address can lead to unintended behaviour such as sending tokens or assigning ownership to a non-existent address. This can result in permanent loss of funds or a contract being locked in an unusable state, introducing denial-of-service risks.

Recommendations

Add explicit zero-address checks, e.g. `require(addr != address(0), "Invalid address")`, to all functions that accept addresses as parameters, particularly when assigning roles, transferring assets, or updating key contract references.

Resolution

In 03aceb4, zero address checks were added to the following functions:

- In `AggLayerGateway.sol`, the `initialize()` function
- In `AggLayerGateway.sol`, the `addPessimisticVKeyRoute()` function
- In `aggchains/AggchainFEP.sol`, the `_initializeAggchain()` function
- In `aggchains/AggchainFEP.sol`, the `transferAggChainManagerRole()` and `transferOptimisticModeManagerRole()` functions.
- In `lib/AggChainBase.sol`, the `constructor` function
- In `lib/AggChainBase.sol`, the `_initializeAggchainBaseAndConsensusBase()` function
- In `sovereignChains/BridgeL2SovereignChain.sol`, the `setBridgeManager()` function
- In `sovereignChains/BridgeL2SovereignChain.sol`, the `_claimWrappedAsset()` function
- In `sovereignChains/GlobalExitRootManagerL2SovereignChain.sol`, the `initialize()` function

AGL3-16	Single-Step Role Transfers
Asset	`contracts/v2/sovereignChains/GlobalExitRootManagerL2SovereignChain.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `globalExitRootUpdater` and `globalExitRootRemover` implement insecure single-step role transfer mechanisms.

If an accidental transfer is made to an uncontrolled address, the role would become unrecoverable. This occurs because only the current role holder has permission to transfer the role. The only way to reclaim the role would be for the uncontrolled address to manually transfer it back, which in most cases would be impossible.

The problematic role-transfer mechanism can be observed in the relevant setters `setGlobalExitRootUpdater()` and `setGlobalExitRootRemover()`.

Losing control over these roles would introduce a severe security risk, as these roles are critical for preventing double spending and fraudulent proofs, as explicitly mentioned at `GlobalExitRootManagerL2SovereignChain.sol` line [19].

Recommendations

Implement a two-step role transfer mechanism or introduce a time delay to mitigate accidental transfers. This delay would allow for corrective action before finalizing the role transfer.

These measures should be taken on similar role-based mechanisms throughout the codebase to prevent similar risks.

Resolution

Two step role transfers were implemented in 03aceb4.

AGL3-17	Lack Of `aggChainData` Length Check
Asset	`contracts/v2/aggchains/AggchainECDSA.sol, AggchainFEP.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In `AggchainECDSA` and `AggchainFEP`, the functions `getAggchainHash()` and `onVerifyPessimistic()` do not perform any length checks on the `aggChainData` byte array prior to decoding.

A malformed or insufficiently sized `aggChainData` input will cause a low level revert during decoding, resulting in ambiguous failure message and wasted gas.

Recommendations

Implement a pre-emptive length check on aggChainData before attempting to decode it. For example:

```
if (aggChainData.length < 34) revert InvalidAggchainData();
```

Resolution

Multiple `aggChainData` length checks were added in 252f3b2.

AGL3-18	Lack Of `proofBytes` Length Check
Asset	`contracts/v2/AggLayerGateway.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `verifyPessimisticProof()` function does not perform any length validation on the `proofBytes` input prior to slicing and parsing. Based on the specification, the `proofBytes` array is expected to follow a specific structure:

1. `proof[0:4]`: 4-byte selector for pp
2. `proof[4:8]`: 4-byte selector for the SP1 verifier
3. `proof[8:]`: remaining bytes representing the proof

Without verifying the total length, slicing operations may revert with a generic "slice out of bounds" error if `proofBytes` is smaller than expected.

While the function will revert regardless, the absence of explicit validation results in uninformative and low-level error messages.

Recommendations

Implement an explicit length check to ensure that `proofBytes` contains at least the minimum number of bytes required to safely perform the slicing and verification steps. For example:

```
if (proofBytes.length < 8) revert InvalidProofBytes();
```

Additionally, if the structure of the trailing proof segment is known or fixed in size, consider validating its length accordingly, to ensure the proof is complete and well formed.

Resolution

A length check for `proofBytes` was added in a3db27d.

AGL3-19	Events Emitted Prior to Successful `initialize()` Call
Asset	`contracts/v2/PolygonRollupManager.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract emits events before calling `IAggchainBase(rollupAddress).initialize(initializeBytesCustomChain)`.

If this initialisation call fails, previously emitted events may incorrectly suggest that the rollup has been successfully deployed and is operational.

Emitting lifecycle or configuration events before successful initialisation can mislead off-chain systems such as indexers, explorers, or monitoring tools into falsely registering the rollup as live.

Recommendations

Reorder logic to ensure that events are only emitted after the `initialize()` call has completed successfully.

Resolution

The logic was reordered in 5053e61.

AGL3-20	Inconsistencies With Error Name
Asset	`aggchains/AggchainFEP.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The code and comments in this file assert that the starting L2 timestamp must be less than or equal to the current time, but the error name is `StartL2TimestampMustBeGreaterThanCurrentTime`, which implies the opposite.

On line [**185**] there is a comment that contradicts the error name:

```
/// @notice starting L2 timestamp must be less than current time
error StartL2TimestampMustBeGreaterThanCurrentTime();
```

On line [**371**] the code logic enforces a less than or equal requirement, which also contradicts the error name:

```
if (_initParams.startingTimestamp > block.timestamp) {
    revert StartL2TimestampMustBeGreaterThanCurrentTime();
}
```

Recommendations

Consider renaming the error to `StartL2TimestampMustNotBeGreaterThanCurrentTime()`, or a more succinct alternative.

Resolution

The error was renamed to `StartL2TimestampMustBeLessThanCurrentTime()` in 952a73e.

AGL3-21	Unnecessary Check At `PolygonRollupManager`
Asset	`PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The check at `PolygonRollupManager` line [**1253**] is redundant:

```
if (rollup.rollupVerifierType != VerifierType.StateTransition)
```

Earlier in the same function at line [**1183**], the following check already prevents this case:

```
// Not for state transition chains
if (rollup.rollupVerifierType == VerifierType.StateTransition) {
    revert StateTransitionChainsNotAllowed();
}
```

Therefore, by the time execution reaches line [**1253**], the `rollup.rollupVerifierType` is guaranteed not to be `StateTransition`, making the later condition always true and unnecessary.

Recommendations

Remove the redundant check at line [**1253**].

Resolution

The development team acknowledged the issue and determined no changes were required at this time.

AGL3-22	Consensus Implementations Storage Layout Could Be Namespaced To Increase Upgrade Safety
Asset	`aggchains/AggchainECDSA.sol aggchains/AggchainFEP.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

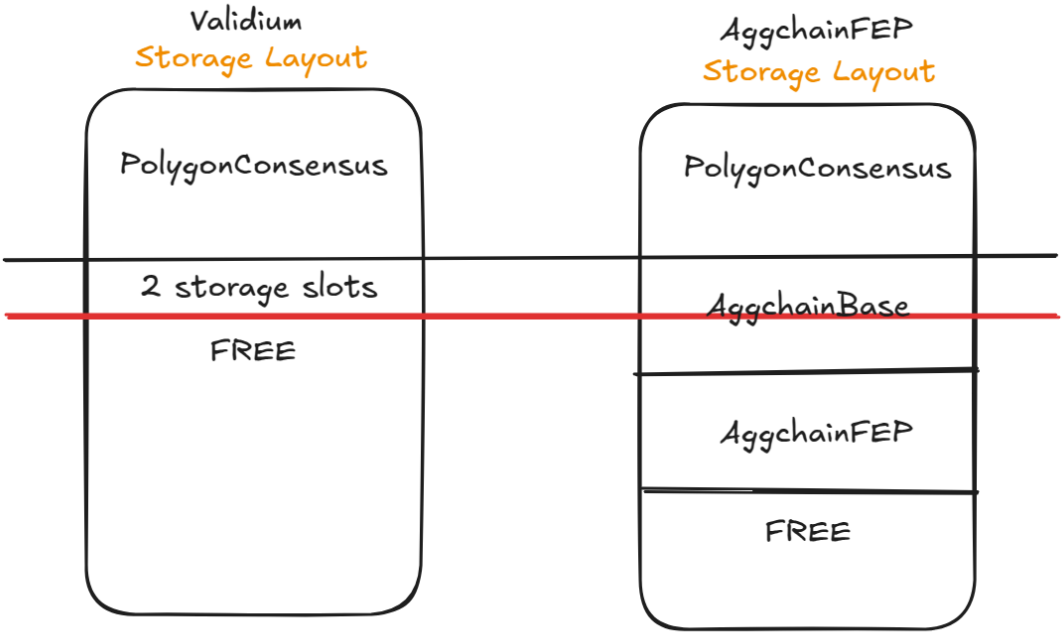
The current architecture for storage layouts for consensus contracts is as follows:

```
// PolygonRollupManager -> RollupProxy -> ConsensusImplementation
```

This setup introduces a risk of overlapping storage layouts between different consensus contracts, which can lead to upgrade incompatibilities.

Here are some upgrade scenarios with realistic motivations:

- `AggchainECDSA -> AggChainFEP` (safe): A centralized ECDSA-based rollup might want to decentralize gradually by switching to FEP.
- `AggchainFEP -> AggchainECDSA` (safe): A decentralized rollup could pivot to a more centralized solution for cost or operational reasons.
- `PolygonValidiumEtrog -> AggChainFEP` (non-breaking but risky): While both inherit from `PolygonConsensusBase`, their subsequent storage layouts diverge. `AggChainFEP` inherits from `AggchainBase`, which includes variables like `pendingVKeyManager`. Meanwhile, `PolygonValidiumEtrog` defines two unique storage variables.



Due to the alignment, `pendingVKeyManager` in `AggchainBase` will end up occupying the same slot as `bool isSequenceWithDataAvailabilityAllowed` in `PolygonValidiumEtrog`. While this happens to be a boolean, resulting in a `0` or `1` value — mapping to `address(0)` or `address(1)` — the risk still exists. If the overridden slot had been a real address, that address could unintentionally gain `vKeyManager` privileges by calling `acceptVKeyManagerRole()`.

This highlights how such overlaps can lead to storage corruption. While current implementations avoid critical conflicts, the Aggchian's intended flexibility and long-term sustainability would benefit from more robust safeguards.

Recommendations

To provide more safe and flexible upgradeability, the consider adopting a namespaced storage layout system for consensus mechanisms, as introduced in OpenZeppelin Contracts v5+.

This strategy gives each consensus implementation a dedicated and isolated storage region, eliminating layout collision risks.

Examples:

- `PolygonConsensusBase` storage layout would start at:

```
keccak256(abi.encode(uint256(keccak256(bytes("Aggchain.PolygonConsensusBase"))) - 1)) &
    ↪ textasciitilde{}bytes32(uint256(0xff))
```

- `AggchainBase` storage layout would start at:

```
keccak256(abi.encode(uint256(keccak256(bytes("Aggchain.AggchainBase"))) - 1)) & textasciitilde{}bytes32(uint256(0xff))
```

Resolution

The development team added legacy storage values to avoid a potential storage collision in 990743b.

AGL3-23	Missing Event Emission At `GlobalExitRootManagerL2SovereignChain` Initialization
Asset	`sovereignChains/GlobalExitRootManagerL2SovereignChain.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The initialization function in this contract updates critical storage variables but does not emit any events when doing so.

However, when these same variables are updated post-initialization, corresponding events are emitted. This inconsistency makes it harder to track and monitor changes.

Recommendations

To ensure consistent and accurate tracking, emit the appropriate events during initialization as well.

```
emit SetGlobalExitRootRemover(_globalExitRootRemover);
emit SetGlobalExitRootUpdater(_globalExitRootUpdater);
```

Resolution

The events were added in ffaecf7.

AGL3-24	Missing Gap On `GlobalExitRootManagerL2SovereignChain`
Asset	`sovereignChains/GlobalExitRootManagerL2SovereignChain.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `GlobalExitRootManagerL2SovereignChain` contract is a proxy and should include storage gaps to prevent collisions in future upgrades. However, it currently lacks such gaps.

The parent contract, `PolygonZkEVMGlobalExitRootL2`, correctly includes a gap (see line [35]), setting a precedent and expectation for safe upgradability.

Omitting the gap increases the risk of storage collisions if future developers extend or upgrade the contract without realizing the layout is unprotected.

Recommendations

Add a storage gap at the end of the contract's state variables. This way future development will clearly see the proxy nature and will be able to expand functionalities in an easier way. For example:

```
// At `GlobalExitRootManagerL2SovereignChain`, insert after the following declaration:

// Value of the removed global exit roots hash chain after last removal
bytes32 public removedGERHashChain;

// Storage gap for future upgrades
uint256[50] private __gap;
```

Resolution

A storage gap was added in b7a3c8b.

AGL3-25	Common Infinite Approval Via `permit()` Is Incompatible With `bridgeAsset()` Calls
Asset	`sovereignChains/BridgeL2SovereignChain.sol PolygonZkEVMBridgeV2.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `bridgeAsset()` function currently lacks support for infinite token approvals via the `permit()` pattern, which is widely adopted in DeFi.

Many ERC20 tokens allow approvals of `uint.max` to enable unlimited spending. If a user tries to use such infinite approval when calling `bridgeAsset()`, the transaction will revert. This behavior aligns with the latest OpenZeppelin ERC20 implementation (see here), which is a standard across most projects.

The issue lies in a check within `BridgeL2SovereignChain.sol` at line [**593**], where it expects `amount` to be equal to `value`, with `value` being the approved amount. If the approval is `uint.max`, this implies the user must hold `uint.max` tokens, which is practically never the case. This check results in the transaction reverting in scenarios where `amount` is not equal to `value`, which is actually the expected behaviour in lots of real world use cases.

The root cause is that `bridgeAsset()` internally calls `_permit()` assuming the transfer amount must always match the approved amount: a notion that doesn't align with modern ERC20 patterns.

> Note: Infinite approvals are common in DEXes and bridges to reduce friction and gas costs. While not the most secure approach, they are widely used.

> Note: The only case where this doesn't revert is if the token supports transfers of `uint.max` to mean that the user's full balance is transferred or if the user actually holds `uint.max` tokens.

Recommendations

Remove the strict enforcement of `value == amount` in the `_permit()` function.

Introduce a flag within the `bytes calldata permitData` to signal whether the approval should be reset to 0 after bridging.

This would allow support for infinite approvals through permits by setting the flag to `false`, while still enabling secure handling where the approval is expected to be consumed entirely and reset.

Conceptual schema:

```
_bridgingLogic();

if (expectedZeroApprovalsAfterBridging) {
    // Standard approvals should set this to `true` and confirm the bridge used exactly the approved amount.
    require(token.readApprovals(address(bridge)) == 0);
}

// If the user intends to allow a remainingApproval > 0, it is assumed to be `uint.max` for infinite approvals.
// Users should be clearly warned in documentation not to set the flag to `false` unless they explicitly are granting infinite
      ↪  approval.
```

Resolution

The strict enforcement of `value == amount` in the `_permit()` function was removed in 8595504.

AGL3-26	Missing Permit Bridging Functionality For Non-native Tokens
Asset	`sovereignChains/BridgeL2SovereignChain.sol PolygonZkEVMBridgeV2.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The code is missing support for `permit()` functionality when bridging `TokenWrapped` tokens.

`TokenWrapped` tokens represent assets not native to the chain and include a `permit()` method in their implementation. However, when using `bridgeAsset()` to bridge these tokens, the system does not offer a way to utilise the permit flow, unlike native tokens, which do support it.

Reference:

- In the `TokenWrapped` contract, line [**70**], the `permit()` method is clearly implemented.
- In `PolygonZkEVMBridgeV2`, lines [**259–264**] handle non-native tokens but omit the `permit()` call.
- However, just a few lines later, line [**268**] handles native tokens and does make use of `permit()`.

```
if (tokenInfo.originTokenAddress != address(0)) {
    // The token is a wrapped token from another network

    _bridgeWrappedAsset(TokenWrapped(token), amount);

    originTokenAddress = tokenInfo.originTokenAddress;
    originNetwork = tokenInfo.originNetwork;
} else {
    // Use permit if any
    if (permitData.length != 0) {
        _permit(token, amount, permitData);
    }
```

Recommendations

Extend the bridging logic to support `permit()` calls when handling non-native `TokenWrapped` tokens. This will ensure consistent and gas-efficient behaviour across asset types.

Resolution

The suggested changes were made in 9a4441f.

AGL3-27	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. **No Length Validation on `_gasTokenMetadata` in Initialiser**

   Related Asset(s): contracts/v2/sovereignChains/BridgeL2SovereignChain.sol

   The `initialize()` function in `BridgeL2SovereignChain` accepts a `_gasTokenMetadata` parameter, but does not validate its length.

   Add length checks and structural validation on `_gasTokenMetadata` to ensure that the expected encoding and minimum byte length are present before parsing.

2. **Missing Events for Key State Changing Actions**

   Related Asset(s): contracts/v2/PolygonRollupManager.sol, contracts/v2/sovereignChains/BridgeL2SovereignChain.sol

   Several key state changing actions across the codebase do not emit corresponding events. This limits traceability and observability for off-chain systems and governance participants.

   - In `BridgeL2SovereignChain`, the initialiser responsible for deploying tokens does not emit an event to signal token deployment or bridge setup completion.
   - In `PolygonRollupManager`, emergency activation and deactivation mechanisms do not emit events when toggled, despite these being high-privilege actions that can significantly alter system behaviour.

   Emit structured, descriptive events for all high-impact actions.

3. **No Hard Cap on Array Length in `setMultipleSovereignTokenAddress()`**

   Related Asset(s): contracts/v2/sovereignChains/BridgeL2SovereignChain.sol

   The `setMultipleSovereignTokenAddress()` function accepts an unbounded `originNetworks` array. Although the function is intended for batch updates, there is currently no restriction on the maximum length of this array. In scenarios where the array is excessively large, the function may consume significant gas or exceed block gas limits.

   Implement a hard cap on the maximum length of the `originNetworks` array to ensure the function remains efficient and predictable.

4. **Missing Check for `amount > 0` in Token Transfers**

   Related Asset(s): contracts/v2/sovereignChains/BridgeL2SovereignChain.sol

   In `_claimWrappedAsset()` the current implementation allows token transfers or operations with `amount == 0`. While this adheres to the ERC-20 standard in theory, some non-standard or historically problematic token contracts do not handle zero-amount transfers correctly.

   Enforce `amount > 0` for transfers to prevent unintended behaviour with non-standard ERC-20 tokens, such as USDT, or tokens with custom hooks or gas optimisations that do not safely handle zero-amount transfers.

5. **Unbounded Reward Payouts Per Verification Transaction**

   Related Asset(s): contracts/v2/PolygonRollupManager.sol

   At line [**1146**], the contract calculates the total reward payout as `calculateRewardPerBatch() * newVerifiedBatches`. There is currently no upper bound on how many batches can be verified in a single transaction, nor on the total payout amount. This creates the potential for unbounded disbursements in a single verification call.

   Introduce an upper limit on the number of batches that can be rewarded per verification transaction, or cap the maximum total reward amount.

6. **Unbounded Loop in `rollbackBatches()`**

   Related Asset(s): contracts/v2/PolygonRollupManager.sol

   The `rollbackBatches()` function includes a while loop on line [**930**] that iteratively rolls back batches from `lastBatchSequenced` down to `targetBatch`. If the difference between these two values is large, the loop may consume excessive gas and revert.

   Introduce a hard cap on the number of iterations permitted in a single call to `rollbackBatches()`.

7. **Invalid Comments**

   Related Asset(s): contracts/v2/AggchainECDSA.sol, AggchainFEP.sol

   The comment on line [**165**] in `AggchainECDSA` describes `AggchainHash` as `AGGCHAIN_TYPE | aggchainVKey | aggchainParams`, whereas, according to spec, it should be `CONSENSUS_TYPE`, not `AGGCHAIN_TYPE`.

   The comment on line [**53**] in `AggchainFEP` incorrectly states `// (...) the first 2 byes of aggchain selector`, whereas it should be "last 2 bytes".

   Update the comments to align with the spec.

8. **Redundant `reinitializer` Check**

   Related Asset(s): contracts/v2/AggchainECDSA.sol, AggchainFEP.sol

   There is an unneccessary check and `revert()` in `AggchainECDSA` on line [**154**]. Currently, with `reinitializer(2)`, this already guarantees the function will only enter for `_initializerVersion < 2`.

   If `reinitializer` version is increased in the future, that would suggest that higher versions are also acceptable, so the same check would already apply.

   Note, the same issue applies to `AggchainFEP` on line [**354**].

   Consider removing the `else` block.

9. **Lack Of Revert Message In `_addLeaf()`**

   Related Asset(s): contracts/v2/lib/DepositContractBase.sol

   Improve debugging clarity by adding a revert message in `DepositContractBase._addLeaf()` on the assert at `DepositContractBase.sol` on line [**88**].

   Currently, if the function fails, it provides no specific description, making it harder to trace issues.

   Add a meaningful revert message to `_addLeaf()` to improve error traceability, e.g.:

   ```
   revert("DepositContractBase: Failed to add leaf");
   ```

10. **Issues In Comments**

    Related Asset(s): aggchains/AggchainECDSA.sol
    sovereignChains/GlobalExitRootManagerL2SovereignChain.sol

    - In `AggchainECDSA.sol` line 162, this comment attached to `getAggchainHash()` may be intended to describe `onVerifyPessimistic()`.

```
* @notice Callback while pessimistic proof is being verified from the rollup manager
```

- In `GlobalExitRootManagerL2SovereignChain.sol` line 148, the words "while the loop" may be better expressed as "while looping" or "within the loop".

Review the relevant comments and update as needed.

11. **Lack of Parameter Validation Of `initializeParams`**

    Related Asset(s): deployment/v2/utils/updateVanillaGenesis.ts

    The `initializeParams` parameter is directly encoded into the transaction data without explicit validation. As a result, a typo or malformed address could silently propagate, leading to an incorrectly initialised contract.

    While the script is expected to be executed in a trusted context, the absence of input validation increases the risk of misconfiguration. Careful review of all input values is strongly recommended during deployment to ensure correctness and prevent unintended contract behaviour.

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. **No Length Validation on `_gasTokenMetadata` in Initialiser**

   The development team acknowledged the issue and determined no changes were required at this time.

2. **Missing Events for Key State Changing Actions**

   Related Asset(s): contracts/v2/PolygonRollupManager.sol, contracts/v2/sovereignChains/BridgeL2SovereignChain.sol

   The development team acknowledged the issue and determined no changes were required at this time.

3. **No Hard Cap on Array Length in `setMultipleSovereignTokenAddress()`**

   The development team acknowledged the issue and determined no changes were required at this time.

4. **Missing Check for `amount > 0` in Token Transfers**

   The development team acknowledged the issue and determined no changes were required at this time.

5. **Unbounded Reward Payouts Per Verification Transaction**

   Related Asset(s): contracts/v2/PolygonRollupManager.sol

   The development team acknowledged the issue and determined no changes were required at this time.

6. **Unbounded Loop in `rollbackBatches()`**

   The development team acknowledged the issue and determined no changes were required at this time.

7. **Invalid Comments**

   The development team determined no changes were required at this time.

8. **Redundant `reinitializer` Check**

   The development team acknowledged the issue and determined no changes were required at this time.

9. **Lack Of Revert Message In `_addLeaf()`**
   The development team acknowledged the issue and determined no changes were required at this time.
10. **Issues In Comments**
    The suggested changes were made in b7a3c8b.
11. **Lack of Parameter Validation Of `initializeParams`**
    The development team acknowledged the issue.

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 3 tests for test/tests-local/GlobalExitRootManagerL2SovereignChainTest.t.sol:GlobalExitRootManagerL2SovereignChainTest
[PASS] test_initialize() (gas: 26559)
[PASS] test_insertGlobalExitRoot() (gas: 73760)
[PASS] test_removeGlobalExitRoots() (gas: 142925)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 10.69ms (299.52µs CPU time)

Ran 2 tests for test/tests-local/AggchainFEP.t.sol:AggchainFEPTest
[PASS] test_initialize() (gas: 43403)
[PASS] test_onVerifyPessimistic() (gas: 83384)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 10.79ms (441.46µs CPU time)

Ran 3 tests for test/tests-local/AggchainECDSA.t.sol:AggchainECDSATest
[PASS] test_aConstant() (gas: 16568)
[PASS] test_initialize() (gas: 42065)
[PASS] test_onVerifyPessimistic() (gas: 24696)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 10.79ms (487.02µs CPU time)

Ran 2 tests for test/tests-local/BridgeL2SovereignChain.t.sol:BridgeL2SovereignChainTest
[PASS] test_initialize() (gas: 33382)
[PASS] test_setMultipleSovereignTokenAddress() (gas: 89304)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 10.93ms (198.27µs CPU time)

Ran 5 tests for test/tests-local/AggLayerGatewayTest.t.sol:AggLayerGatewayTest
[PASS] test_addDefaultAggchainVKey() (gas: 47426)
[PASS] test_addPessimisticVKeyRoute() (gas: 76015)
[PASS] test_freezePessimisticVKeyRoute() (gas: 147316)
[PASS] test_updateDefaultAggchainVKey() (gas: 52696)
[PASS] test_verifyPessimisticProof() (gas: 178366)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 11.08ms (292.33µs CPU time)

Ran 5 tests for test/tests-local/PolygonRollupManager.t.sol:PolygonRollupManagerTest
[PASS] test_addExistingRollup() (gas: 232711)
[PASS] test_addNewRollupType() (gas: 135269)
[PASS] test_attachAggchainToAL() (gas: 3397044)
[PASS] test_obsoleteRollupType() (gas: 119660)
[PASS] test_updateRollup() (gas: 5957277)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 12.46ms (1.71ms CPU time)

Ran 6 test suites in 16.40ms (66.73ms CPU time): 20 tests passed, 0 failed, 0 skipped (20 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.

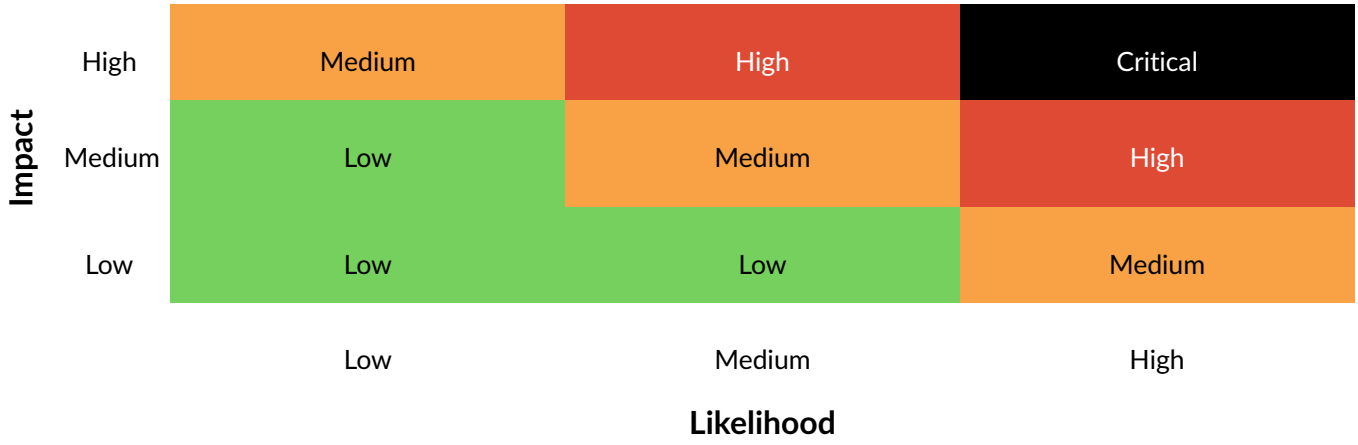


Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].