# Polygon Agg Security Review


**Auditors**

Cergyk, Lead Security Researcher

Shotes, Lead Security Researcher

Zigtur, Lead Security Researcher

**Report prepared by:** Lucas Goiriz



September 17, 2025

# Contents

# 1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

# 2 Introduction

Agglayer is a cross-chain settlement layer that connects the liquidity and users of any blockchain for fast, low cost interoperability and growth.

*Disclaimer*: This security review does not guarantee against a hack. It is a snapshot in time of Polygon Agg according to the specific commit. Any modifications to the code will require a new security review.

# 3 Risk classification

| Severity level | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: high** | Critical | High | Medium |
| **Likelihood: medium** | High | Medium | Low |
| **Likelihood: low** | Medium | Low | Low |

## 3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

## 3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

## 3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

# 4 Executive Summary

Over the course of 47 days in total, Polygon engaged with Spearbit to review the following protocols:

- agglayer/agglayer
- agglayer/provers
- agglayer/aggkit
- agglayer/agglayer-contracts

In this period of time a total of **48** issues were found.

**Summary**

| | |
|---|---|
| **Project Name** | Polygon |
| **Repositories** | agglayer/agglayer, agglayer/provers, agglayer/aggkit, agglayer/agglayer-contracts |
| **Commits** | 933753...b4f1 (tag `v0.3.0-rc.17`), cf8661a5...0b76 (tag `v0.1.0-rc.23`), 2d68cb...f917 (tag `v0.3.0-beta3`), 07b40c...4d43 (branch `v10.1.0-rc.4`) |
| **Type of Project** | Bridge, Aggregator |
| **Audit Timeline** | May 21st to Jul 7th |

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 7 | 7 | 0 |
| Medium Risk | 2 | 1 | 1 |
| Low Risk | 16 | 8 | 8 |
| Gas Optimizations | 0 | 0 | 0 |
| Informational | 23 | 15 | 8 |
| **Total** | **48** | **31** | **17** |

# 5 Findings

## 5.1 High Risk

### 5.1.1 Permissionless certificate submissions can break AggchainFEP chain's aggsender

**Severity:** High Risk

**Context:** aggsender.go#L232-L252, aggsender_initial_state.go#L154-L160, lib.rs#L442-L455

**Summary:** A malicious party can submit valid certificates to the Agglayer for an `AggchainFEP` chain on behalf of the AggSender of this chain. This malicious action will desynchronize the `AggSender` from the Agglayer node, leading to a denial of service.

**Finding Description:** In the Agglayer architecture, the `AggSender` is part of the AggKit component. It is the service that sends the chain certificates to the Agglayer node. `AggchainFEP` certificates can be submitted to the Agglayer node by anyone. There is no access control check (e.g. signature). These certificates embed an Aggchain ZK proof to ensure the correctness of the state transition. This breaks the assumption that the `AggSender` is the only component sending certificates. However, the `AggSender` service is not able to synchronize with external certificates that it didn't generate. Then, when another party submits a certificate, the `AggSender` will not be able to build certificates on top of the state validated with this external certificate.

**Impact Explanation:** Medium: Denial of service of AggchainFEP chains. The AggSender is desynchronized from the Agglayer and will not be able to submit new certificates.

**Likelihood Explanation:** High: A malicious party can submit a valid certificate through the `SubmitCertificate` gRPC endpoint.

Exploit scenario & Code snippets: The `SubmitCertificate` gRPC endpoint can be used by anyone. Then, no signer is expected for `AggchainData::Generic` used by `AggchainFEP` chains.

```
pub fn signer(&self) -> Result<Option<Address>, SignatureError> {
    match self.aggchain_data {
        AggchainData::ECDSA { signature } => {
            // retrieve signer
            let version = CommitmentVersion::V2;
            let combined_hash = SignatureCommitmentValues::from(self).commitment(version);

            signature
                .recover_address_from_prehash(&B256::new(combined_hash.0))
                .map(Some)
        }
        _ => Ok(None), // @audit no signer expected for AggchainFEP
    }
}
```

The `AggSender` has no functionality to sync the latest certificate on `AggLayer` with its local state. The `Initial-Status.Process` shows that the `AggSender` will not be able to recover from the desynchronization when the latest local certificate is not the latest `AggLayer` one.

```
// Process checks the last certificates from agglayer vs local certificates and returns the action to
↪  take
func (i *InitialStatus) Process() (*InitialStatusResult, error) {

    // ...

    // CASE 4: AggSender and AggLayer are not on the same page
    // note: we don't need to check individual fields of the certificate
    // because CertificateID is a hash of all the fields
    if localLastCert.CertificateID != aggLayerLastCert.CertificateID { // @audit fail here when the
    ↪  AggLayer certificate has higher height than the local one
        return nil, fmt.Errorf("recovery: Local certificate:\n %s \n is different from agglayer
        ↪  certificate:\n %s",
            localLastCert.String(), aggLayerLastCert.String())
    }
```

**Recommendation:** There are multiple ways to fix this issue.

- Fix 1: Implement Access Control: An ECDSA signature verification could be added to AggchainFEP certificates. This will ensure that only the expected AggSender component is submitting certificates. This fix will break the permissionless capabilities.
- Fix 2: `AggSender` Synchronization: The `AggSender` component could synchronize external certificates submitted to the Agglayer node and already settled.

**Polygon:** Fixed in protocol-research issue 141.

**Spearbit:** Fixed. The implementation was done in PR 808 of the agglayer repository. The trusted sequencer must sign the certificates for them to be accepter by the agglayer.

#### 5.1.2 Malicious token can DOS Aggkit by forcing to send an unappliable certificate

**Severity:** High Risk

**Context:** lib.rs#L587-L595

**Description:** The agglayer is using `uint256` checked math to compute the amount that is transferred for a token during the imported_bridge_exits amount evaluation:

- agglayer/crates/agglayer-types/src/lib.rs#L589-L595:

```
  for imported_bridge_exit in imported_bridge_exits {
      let token = imported_bridge_exit.bridge_exit.amount_token_info();
      new_balances.insert(
          token,
          new_balances[&token]
              //@audit checked_add on Uint256
              .checked_add(imported_bridge_exit.bridge_exit.amount)
              .ok_or(Error::BalanceOverflow(token))?,
      );
  }
```

Unfortunately, a specifically crafted malicious token can be used to always make a certificate application fail, which will halt the certificate settlement process for the given L2;

1. The attacker deploys a token $PSN on mainnet, over which he has total control.
2. The attacker bridges for `type(uint256).max` to `PolygonZkEVMBridgeV2`:

- contracts/v2/PolygonZkEVMBridgeV2.sol:

```
{
    // In order to support fee tokens check the amount received, not the transferred
    uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(
        address(this)
    );
    IERC20Upgradeable(token).safeTransferFrom(
        msg.sender,
        address(this),
        amount
    );
    //@audit balanceAfter == type(uint256).max
    uint256 balanceAfter = IERC20Upgradeable(token).balanceOf(
        address(this)
    );

    // Override leafAmount with the received amount
    leafAmount = balanceAfter - balanceBefore;

    originTokenAddress = token;
    originNetwork = networkID;
}
```

3. The attacker has to reset the balance of `PolygonZkEVMBridgeV2` to avoid reverting the next transfer, and bridge `type(uint256).max` of $PSN again.
4. On target L2, the attacker has to do the following in a single transaction:
    - Claim `type(uint256).max` (wrapped token is deployed and minted).
    - Bridge back `type(uint256).max` (wrapped token is burnt).
    - Claim `type(uint256).max` (wrapped token is minted).
    - Bridge back `type(uint256).max` (wrapped token is burnt).

This makes sure that the block has `type(uint256).max * 2` of claims for the token $PSN. This will fail when the certificate is applied in agglayer/crates/agglayer-types/src/lib.rs

**Recommendation:** Use `U512` checked math to compute the sum of imported bridge exits:

```
 for imported_bridge_exit in imported_bridge_exits_non_native {
     let token = imported_bridge_exit.bridge_exit.amount_token_info();
     new_balances.insert(
         token,
         new_balances[&token]
-            .checked_add(imported_bridge_exit.bridge_exit.amount)
+            .checked_add(U512::from(imported_bridge_exit.bridge_exit.amount))
             .ok_or(Error::BalanceOverflow(token))?,
     );
 }
```

**Polygon:** Fixed in agglayer PR 849.

**Spearbit:** Fixed. The `U512` type is now used as recommended.

### 5.1.3 Reverted sub-calls on a chain leads to Agglayer denial of service

**Severity:** High Risk

**Context:** downloader.go#L147-L178, downloader.go#L187-L193

**Summary:** The AggKit parses claim data from reverted sub-calls while it should not. This leads to reading invalid data that can deny Aggchain proof generation and panic the AggKit node.

**Description:** The AggKit node currently parses the calldata of calls made to the bridge address in a transaction when a claim event was found in this specific transaction. It retrieves multiple fields from the calldata with the expected global index.

According the `debug_traceTransaction` API, all the sub-calls of the traced transaction are returned. This includes the sub-calls that reverted, for which the `error` and `revertReason` fields are set.

However, the AggKit `setClaimCalldata` function does not ignore the reverted sub-calls. These may contain invalid calldata that the AggKit node will parse and use to generate the Aggchain proof. This proof generation will fail.

```
func setClaimCalldata(client EthClienter, bridge common.Address, txHash common.Hash, claim *Claim)
↪  error {
    c := &call{}
    err := client.Client().Call(c, "debug_traceTransaction", txHash, tracerCfg{Tracer: "callTracer"})
    // ...

    // find the claim linked to the event using DFS
    callStack := stack.New()
    callStack.Push(*c)
    for {
        if callStack.Len() == 0 {
            break
        }

        currentCallInterface := callStack.Pop() // @audit: can pop reverted sub-call
        currentCall, ok := currentCallInterface.(call)
        if !ok {
            return fmt.Errorf("unexpected type for 'currentCall'. Expected 'call', got '%T'",
            ↪  currentCallInterface)
        }
        // @audit: reverted sub-calls should be ignored here

        if currentCall.To == bridge {
            // @audit: process the calldata
            // ...
        }
        for _, c := range currentCall.Calls {
            callStack.Push(c) // @audit: reverting sub-calls are pushed on the stack
        }
    }
    return db.ErrNotFound
}
```

**Impact Explanation: Medium**: Denial of service of the Agglayer service for the chain on which the claim is made. For example, the AggKit node will panic due to out-of-bound when a parsed reverted sub-call doesn't have calldata (i.e. zero length) at `setClaimIfFoundOnInput`:

```
func setClaimIfFoundOnInput(input []byte, claim *Claim) (bool, error) {
    // ...
    methodID := input[:4] // @audit: will panic with `runtime error: slice bounds out of range`
```

**Likelihood Explanation:** High: Exploiting this issue only requires deploying a smart contract with reverting sub-calls.

**Proof of Concept:**

Run a GETH node:

```
cd aggkit/bridgesync
docker compose up
```

Create a forge repository and import the following in `src/Contracts.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract PolygonBridge {
    uint256 public counter;

    event ClaimProcessed(uint256 value, uint256 newCounter);

    function claim(uint256 value) external {
        if (value == 0) {
            revert("AGGLAYER IS VULNERABLE");
        }
        else {
            counter++;
            emit ClaimProcessed(value, counter);
        }
    }
}

contract TestExternalCalls {
    PolygonBridge public polygonBridge;

    event CallResult(string callType, bool success, bytes returnData);

    constructor(address _polygonBridge) {
        polygonBridge = PolygonBridge(_polygonBridge);
    }

    function makeTwoCalls() external returns (bool firstCallSuccess, bool secondCallSuccess) {
        // First call - this will revert but we catch it
        try polygonBridge.claim(0) {
            firstCallSuccess = true;
            emit CallResult("First call (value=0)", true, "");
        } catch Error(string memory reason) {
            firstCallSuccess = false;
            emit CallResult("First call (value=0)", false, bytes(reason));
        }
        // Second call - this will succeed
        try polygonBridge.claim(1) {
            secondCallSuccess = true;
            emit CallResult("Second call (value=1)", true, "");
        } catch Error(string memory reason) {
            secondCallSuccess = false;
            emit CallResult("Second call (value=1)", false, bytes(reason));
        }
        return (firstCallSuccess, secondCallSuccess);
    }
}
```

Then, run the following bash script:

```
#!/bin/bash
PRIVATE_KEY=0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
RPC_URL=http://127.0.0.1:8545

# 1. Deploy PolygonBridge contract
```

```
echo " Deploying PolygonBridge..."
POLYGON_BRIDGE=$(forge create src/Contracts.sol:PolygonBridge \
    --rpc-url $RPC_URL --broadcast \
    --private-key $PRIVATE_KEY \
    --json | jq -r '.deployedTo')

# 2. Deploy TestExternalCalls contract with PolygonBridge address
echo " Deploying TestExternalCalls..."
TEST_CONTRACT=$(forge create src/Contracts.sol:TestExternalCalls \
    --rpc-url $RPC_URL --broadcast --json \
    --private-key $PRIVATE_KEY \
    --constructor-args $POLYGON_BRIDGE \
   | jq -r '.deployedTo')

echo ""
echo " All contracts deployed!"
echo " Contract addresses:"
echo "  PolygonBridge:     $POLYGON_BRIDGE"
echo "  TestExternalCalls: $TEST_CONTRACT"
echo ""

TX_HASH_TRYCATCH=$(cast send $TEST_CONTRACT \
    "makeTwoCalls()" \
    --rpc-url $RPC_URL \
    --private-key $PRIVATE_KEY \
    --json | jq -r '.transactionHash')

echo "Transaction hash (try/catch): $TX_HASH_TRYCATCH"
echo ""

echo " Transaction to trace:"
echo "makeTwoCalls(): $TX_HASH_TRYCATCH"
echo ""
echo " Trace with debug_traceTransaction:"
echo "cast rpc debug_traceTransaction $TX_HASH_TRYCATCH '{\"tracer\": \"callTracer\"}' --rpc-url
↪  \$RPC_URL"
echo ""
echo "Expected behavior:"
echo "   TestExternalCalls.makeTwoCalls() (SUCCESS)"
echo "    PolygonBridge.claim(0) (REVERTED: AGGLAYER IS VULNERABLE)"
echo "    PolygonBridge.claim(1) (SUCCESS)"


cast rpc debug_traceTransaction $TX_HASH_TRYCATCH '{"tracer": "callTracer"}' --rpc-url $RPC_URL | jq
```

**Results:**

```
 Deploying PolygonBridge...
 Deploying TestExternalCalls...

 All contracts deployed!
 Contract addresses:
  PolygonBridge:     0x4b6aB5F819A515382B0dEB6935D793817bB4af28
  TestExternalCalls: 0xCace1b78160AE76398F486c8a18044da0d66d86D

Transaction hash (try/catch): 0x67722a8c778539d316b0f30549e48d1f6de6a865b1ee246099f6f56b883725a1

 Transaction to trace:
makeTwoCalls(): 0x67722a8c778539d316b0f30549e48d1f6de6a865b1ee246099f6f56b883725a1

 Trace with debug_traceTransaction:
```

```
cast rpc debug_traceTransaction 0x67722a8c778539d316b0f30549e48d1f6de6a865b1ee246099f6f56b883725a1
↪  '{"tracer": "callTracer"}' --rpc-url $RPC_URL

Expected behavior:
   TestExternalCalls.makeTwoCalls() (SUCCESS)
    PolygonBridge.claim(0) (REVERTED: AGGLAYER IS VULNERABLE)
    PolygonBridge.claim(1) (SUCCESS)
{
  "from": "0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266",
  "gas": "0xeb3c",
  "gasUsed": "0xe809",
  "to": "0xcace1b78160ae76398f486c8a18044da0d66d86d",
  "input": "0x0ac541b8",
  "output": "0x0000000000000000000000000000000000000000000000000000000000000000
  ↪  0000000000000000000000000000000000000001",
  "calls": [
    {
      "from": "0xcace1b78160ae76398f486c8a18044da0d66d86d",
      "gas": "0x8204",
      "gasUsed": "0x2b6",
      "to": "0x4b6ab5f819a515382b0deb6935d793817bb4af28",
      "input": "0x379607f500000000000000000000000000000000000000000000000000000000000000000",
      "output": "0x08c379a0000000000000000000000000000000000000000000000000000000000000020000000000000
      ↪  0000000000000000000000000000000000000000000000000164147474c4159455220495320565554c4e455241424
      ↪  c450000000000000000000",
      "error": "execution reverted",
      "revertReason": "AGGLAYER IS VULNERABLE",
      "value": "0x0",
      "type": "CALL"
    },
    {
      "from": "0xcace1b78160ae76398f486c8a18044da0d66d86d",
      "gas": "0x6cef",
      "gasUsed": "0x5efa",
      "to": "0x4b6ab5f819a515382b0deb6935d793817bb4af28",
      "input": "0x379607f500000000000000000000000000000000000000000000000000000000000000001",
      "value": "0x0",
      "type": "CALL"
    }
  ],
  "value": "0x0",
  "type": "CALL"
}
```

The output shows the reverted sub-calls that the AggKit node will parse.

**Recommendation:** `setClaimCalldata` must ignore calls that have the `error` field set. This will ensure that sub-calls with errors are not parsed for calldata. Moreover, `setClaimIfFoundOnInput` must check that the input length is greater than or equal to 4 before slicing it to avoid panic.

**Polygon:** Fixed in aggkit PR 656.

**Spearbit:** Fixed. The `error` JSON field is decoded. When it is set in a sub-call, this sub-call is ignored as it has reverted.

#### 5.1.4 Mismatch in `GlobalIndex` parsing forces invalid proofs in AggKit

**Severity:** High Risk

**Context:** processor.go#L331-L350

**Summary:** The AggKit and Agglayer lose information about unused bits from the original U256 Global Index value

during parsing and reconstruction. The lost unused bits can lead to a discrepancy in Global Index hash values, which would result in being unable to generate or verify an aggchain proof.

**Finding Description:** In both AggSender and Agglayer, the Global Index value is parsed out from a claim into its components - mainnetFlag, rollupID, and leafIdx. Importantly, the mainnetFlag is a boolean and cannot contain information about all of the unused higher bits. When reconstructing the Global Index, non-zero bits from the unused portion of the original U256 global index - the most significant bits above the mainnetFlag bit and the rollup ID bits if the mainnetFlag is true - are lost.

For AggKit, see aggkit/bridgesync/processor.go#L336-L345:

```
//@audit mainnetFlag is true, so rollupIndex goes unused
if mainnetFlag {
    //@audit since mainnetFlag is a bool, none of the higher bytes are preserved.
    globalIndexBytes = append(globalIndexBytes, big.NewInt(1).Bytes()...)
    ri := big.NewInt(0).FillBytes(buf[:])
    globalIndexBytes = append(globalIndexBytes, ri...)
} else {
    ri := big.NewInt(0).SetUint64(uint64(rollupIndex)).FillBytes(buf[:])
    globalIndexBytes = append(globalIndexBytes, ri...)
}
leri := big.NewInt(0).SetUint64(uint64(localExitRootIndex)).FillBytes(buf[:])
globalIndexBytes = append(globalIndexBytes, leri...)
```

For AggLayer, see unified-bridge/src/global_index.rs#L122-L135:

```
fn from(value: GlobalIndex) -> Self {
    let mut bytes = [0u8; 32];

    let leaf_bytes = value.leaf_index.to_le_bytes();
    bytes[0..4].copy_from_slice(&leaf_bytes);

    let rollup_bytes = value.rollup_index.to_le_bytes();
    bytes[4..8].copy_from_slice(&rollup_bytes);

    //@audit since mainnetFlag is a bool, none of the higher bytes are preserved.
    if value.mainnet_flag {
        bytes[8] |= 0x01;
    }

    U256::from_le_bytes(bytes)
}
```

In the `PolygonZkEVMBridgeV2.sol` contract, the claim functions do not assert the unused bits to 0. This creates a discrepancy between the Global Index value emitted on chain and the Global Index value reconstructed in AggSender/Agglayer when the Global Index contains the mainnetFlag. For example, a global index claim of 0x1FFFFFFFFFFFFFFF will be:

- On Chain: 0xAAAA1FFFFFFFFFFFFFFF (higher bits ignored, rollup ID ignored).
- AggSender: 0x100000000FFFFFFF (higher bits zerod out, rollup ID zeroed out).

In the L2 Sovereign Chain bridge work flow, the Global Index values from on-chain are hashed into the `BridgeL2SovereignChain::claimedGlobalIndexHashChain`. This hash chain is verified in the aggchain proof program, comparing the hash chain from on-chain to the reconstructed hash chain using the Global Index values supplied in the proof inputs.

- See provers/crates/aggchain-proof-core/src/bridge/mod.rs#L223-L246:

```
// Verify the hash chain on claimed global index
let hash_chain_type = HashChainType::ClaimedGlobalIndex;
let (prev_hash_chain, new_hash_chain): (Digest, Digest) = self
    .fetch_hash_chains(
        hash_chain_type,
        bridge_address,
        BridgeL2SovereignChain::claimedGlobalIndexHashChainCall {},
    )
    .map(|(prev, new)| (prev.hashChain.0.into(), new.hashChain.0.into()))?;

//@audit bridge_exits_claimed contains global index input from AggSender
let claims: Vec<Digest> = self
    .bridge_witness
    .bridge_exits_claimed
    .iter()
    .map(|&idx| idx.commitment())
    .collect();

self.validate_hash_chain(&claims, prev_hash_chain, new_hash_chain, hash_chain_type)?;
```

The discrepancy between the Global Index values on-chain and in AggKit means that the hash chain reconstruction will not be valid, and the proof generation will fail. The discrepancy in Agglayer means that verification of the proof would also fail.

**Impact Explanation:** Medium: AggSender will become unable to issue any certificates for the L2 sovereign chain on which the invalid claim is made.

**Likelihood Explanation:** High: Exploiting this issue only requires bridging an asset from any chain to an L2 sovereign chain.

**Proof of Concept:**

1. Call `bridgeAsset()` on a bridge `PolygonZkEVMBridgeV2`. The `destinationNetwork` must be the L2 sovereign chain network.
2. Wait for the asset to become claimable on the L2 sovereign chain.
3. Call `claimAsset()` on the L2 sovereign chain bridge `BridgeL2SovereignChain`. Set any bits above the `mainnetFlag` portion of the `globalIndex` parameter to any non-zero value.
4. `AggSender` for the L2 sovereign chain will not be able to process this claim.

**Recommendation:** There are a few different options available as a solution to this issue. There can even be a mix of the options to form a complete solution. For example, only enforcing unused bits in the higher bytes of globalIndex, while forcing AggKit and AggSender to preserve data in the rollup ID bitfield.

- Option 1: Enforce unused bits to 0 in the `PolygonZkEVMBridgeV2.sol` contract. This would be the most complete solution, as it is strict with the input requirements for the whole system. For example, the following logic could be added to `_verifyLeaf` to verify the global index:

```
uint256 internal constant _GLOBAL_INDEX_MAINNET_FLAG = 2 ** 64;
uint256 internal constant _LEAF_INDEX_MASK = 0xFFFFFFFF;
uint256 internal constant _ROLLUP_ID_MASK = 0xFFFFFFFF00000000;
uint256 internal constant _ALLOWED_MASK = _GLOBAL_INDEX_MAINNET_FLAG | _LEAF_INDEX_MASK |
↪  _ROLLUP_ID_MASK;

function _verifyLeaf(
    // ...,
    uint256 globalIndex,
    // ...,
) internal virtual {
    // ...

    // Verify unused bits are zero
    if (globalIndex & ~_ALLOWED_MASK != 0) {
            revert GlobalIndexInvalid();
    }

    if (globalIndex & _GLOBAL_INDEX_MAINNET_FLAG != 0) {
        // Verify rollup ID bits are zero while mainnet flag is set
        if (globalIndex & _ROLLUP_ID_MASK != 0) {
            revert GlobalIndexInvalid();
        }
    }
    // ...
}
```

- Option 2: Drop all unused bits during the hashing of the globalIndex value in the `claimedGlobalIndexHashChain` in `BridgeL2SovereignChain.sol`. This preserves the current behavior of the claim verification and allows aggkit and agglayer to safely drop unused bits.

```
uint256 internal constant _GLOBAL_INDEX_MAINNET_FLAG = 2 ** 64;
uint256 internal constant _LEAF_INDEX_MASK = 0xFFFFFFFF;
uint256 internal constant _ROLLUP_ID_MASK = 0xFFFFFFFF00000000;
uint256 internal constant _ALLOWED_MASK = _GLOBAL_INDEX_MAINNET_FLAG | _LEAF_INDEX_MASK |
↪  _ROLLUP_ID_MASK;

function _verifyLeafBridge(
    // ...,
    globalIndex,
    // ...,
) internal override {
    // ...
    // globalIndex hash chain input must not contain unused bytes
    uint256 cleanGlobalIndex = globalIndex & _ALLOWED_MASK;
    if (globalIndex & _GLOBAL_INDEX_MAINNET_FLAG != 0){
        // Remove rollup ID bytes when mainnet flag is set
        cleanGlobalIndex = cleanGlobalIndex & ~_ROLLUP_ID_MASK;
    }

    // Update claimedGlobalIndexHashChain
    claimedGlobalIndexHashChain = Hashes.efficientKeccak256(
        claimedGlobalIndexHashChain,
        Hashes.efficientKeccak256(bytes32(cleanGlobalIndex), leafValue)
    );
    // ...
}
```

- Option 3: Ensure the preservation of the data in the entire U256 global index across AggKit and Agglayer. This will require some refactoring of the `globalIndex` struct usage across both codebases and is a bit more

involved.

**Polygon:** Fixed in agglayer-contracts PR 489.

**Spearbit:** Fix verified. The verification of the reconstructed `globalIndex` is a clean, succinct solution.

#### 5.1.5 Making a nested `claimMessage` with reentrancy call will DOS Aggkit on chain using Generic AggChain Proofs

**Severity:** High Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** The legacy pessimistic proofs check that imported bridge exits are claims which have a proof (`globalIndex` inclusion in `L1InfoTreeLeaf`). In addition to that, the new Generic Aggchain Proofs also check that all claims performed on the L2 contract are included using the `claimedGlobalIndexHashChain`.

- BridgeL2SovereignChain.sol#L1093-L1097:

```
// Update claimedGlobalIndexHashChain
claimedGlobalIndexHashChain = Hashes.efficientKeccak256(
    claimedGlobalIndexHashChain,
    Hashes.efficientKeccak256(bytes32(globalIndex), leafValue)
);
```

However, due to the claimMessage endpoint allowing for reentrancy:

- PolygonZkEVMBridgeV2.sol#L725-L737:

```
function claimMessage(
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata smtProofLocalExitRoot,
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata smtProofRollupExitRoot,
    uint256 globalIndex,
    bytes32 mainnetExitRoot,
    bytes32 rollupExitRoot,
    uint32 originNetwork,
    address originAddress,
    uint32 destinationNetwork,
    address destinationAddress,
    uint256 amount,
    bytes calldata metadata
//@audit no nonReentrant modifier
) external ifNotEmergencyState {
```

Any user making a reentrant call calling either `claimMessage` or `claimAsset` will make the contract emit the logs in the reverse order as the hashes are added to the chain:

```
function claimMessage(
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata smtProofLocalExitRoot,
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata smtProofRollupExitRoot,
    uint256 globalIndex,
    bytes32 mainnetExitRoot,
    bytes32 rollupExitRoot,
    uint32 originNetwork,
    address originAddress,
    uint32 destinationNetwork,
    address destinationAddress,
    uint256 amount,
    bytes calldata metadata
) external ifNotEmergencyState {
    // Destination network must be this networkID
    if (destinationNetwork != networkID) {
        revert DestinationNetworkInvalid();
```

```
    }

    // Verify leaf exist and it does not have been claimed
    //@audit claim is added to hashChain inside this function
    _verifyLeafBridge(
        smtProofLocalExitRoot,
        smtProofRollupExitRoot,
        globalIndex,
        mainnetExitRoot,
        rollupExitRoot,
        _LEAF_TYPE_MESSAGE,
        originNetwork,
        originAddress,
        destinationNetwork,
        destinationAddress,
        amount,
        keccak256(metadata)
    );

    // Execute message
    bool success;
    if (address(WETHToken) == address(0)) {
        // Native token is ether
        // Transfer ether
        /* solhint-disable avoid-low-level-calls */
        //@audit external call is made, this call can reenter on claimMessage or claimAsset
        (success, ) = destinationAddress.call{value: amount}(
            abi.encodeCall(
                IBridgeMessageReceiver.onMessageReceived,
                (originAddress, originNetwork, metadata)
            )
        );
    } else {
        // ... //@audit no native value call logic
    }

    if (!success) {
        revert MessageFailed();
    }

    //@audit the event is emitted for the aggkit to be picked up
    emit ClaimEvent(
        globalIndex,
        originNetwork,
        originAddress,
        destinationAddress,
        amount
    );
}
```

As we can see, in the case of a nested `claimMessage` call, the sequence of operations goes as follows:

```
1. Outer claim hash is added to hash chain
**Enter the external call**
    2. Inner claim hash is added to hash chain
    3. Inner claim log is emitted
**Exit the external call**
4. Outer claim log is emitted
```

As a result, the aggkit will consider the inner claim happened before the outer claim because of the log order; However, attempting to reconstruct the `claimedGlobalIndexHashChain` will yield the wrong value, and the proof

cannot be verified.

**Recommendation:** Either restrict reentrancy on the `claimMessage` endpoint, or append the claim hash to the `claimedGlobalIndexHashChain` after the external call has been made.

**Polygon:** Fixed in agglayer-contracts PR 489.

**Spearbit:** Fix verified.

### 5.1.6 Unchecked upper bits in global index can cause Aggkit to stop processing L2 blocks

**Severity:** High Risk

**Context:** processor.go#L352-L365

**Description:** When claiming through the `contracts/v2/PolygonZkEVMBridgeV2.sol`, the user must provide a `globalIndex` 256-bit value of which only the lower 65 bits are used to determine the path in the `L1InfoTree`:

- PolygonZkEVMBridgeV2.sol#L917-L983:

```
function _verifyLeaf(
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata smtProofLocalExitRoot,
    bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] calldata smtProofRollupExitRoot,
    uint256 globalIndex,
    bytes32 mainnetExitRoot,
    bytes32 rollupExitRoot,
    bytes32 leafValue
) internal virtual {
    // Check blockhash where the global exit root was set
    // Note that previous timestamps were set, since in only checked if != 0 it's ok
    uint256 blockHashGlobalExitRoot = globalExitRootManager
        .globalExitRootMap(
            GlobalExitRootLib.calculateGlobalExitRoot(
                mainnetExitRoot,
                rollupExitRoot
            )
        );

    // check that this global exit root exist
    if (blockHashGlobalExitRoot == 0) {
        revert GlobalExitRootInvalid();
    }

    uint32 leafIndex;
    uint32 sourceBridgeNetwork;

    // Get origin network from global index
    //@audit globalIndex higher bits are silently ignored
    if (globalIndex & _GLOBAL_INDEX_MAINNET_FLAG != 0) { //@audit _GLOBAL_INDEX_MAINNET_FLAG ==
    ↪  65
        // the network is mainnet, therefore sourceBridgeNetwork is 0

        // Last 32 bits are leafIndex
        leafIndex = uint32(globalIndex);

        if (
            !verifyMerkleProof(
                leafValue,
                smtProofLocalExitRoot,
                leafIndex,
                mainnetExitRoot
            )
        ) {
```

```
            revert InvalidSmtProof();
        }
    } else {
        // the network is a rollup, therefore sourceBridgeNetwork must be decoded
        //@audit globalIndex higher bits are silently ignored
        uint32 indexRollup = uint32(globalIndex >> 32);
        sourceBridgeNetwork = indexRollup + 1;

        // Last 32 bits are leafIndex
        //@audit globalIndex higher bits are silently ignored
        leafIndex = uint32(globalIndex);

        // Verify merkle proof against rollup exit root
        if (
            !verifyMerkleProof(
                calculateRoot(leafValue, smtProofLocalExitRoot, leafIndex),
                smtProofRollupExitRoot,
                indexRollup,
                rollupExitRoot
            )
        ) {
            revert InvalidSmtProof();
        }
    }

    // Set and check nullifier
    _setAndCheckClaimed(leafIndex, sourceBridgeNetwork);
}
```

However, we can see in the contract that the higher unused bits of the `globalIndex` are silently ignored, and the call can succeed if the `globalIndex` is more than 9 bytes.

But when converting the Claim to an imported bridge exit in aggkit, the following check is made:

- aggkit/bridgesync/processor.go#L352-L365:

```
// Decodes global index to its three parts:
// 1. mainnetFlag - first byte
// 2. rollupIndex - next 4 bytes
// 3. localExitRootIndex - last 4 bytes
// NOTE - mainnet flag is not in the global index bytes if it is false
// NOTE - rollup index is 0 if mainnet flag is true
// NOTE - rollup index is not in the global index bytes if mainnet flag is false and rollup index
↪  is 0
func DecodeGlobalIndex(globalIndex *big.Int) (mainnetFlag bool,
    rollupIndex uint32, localExitRootIndex uint32, err error) {
    globalIndexBytes := globalIndex.Bytes()
    l := len(globalIndexBytes)
    if l > globalIndexMaxSize { //@audit globalIndexMaxSize == 9
        return false, 0, 0, errors.New("invalid global index length")
    }
```

Which means the decoding will fail, and the Aggsender will stop processing further blocks to include them in certificates for the agglayer.

**Impact:** Medium, certificate sending is blocked for the time needed to fix the issue in aggkit and agglayer.

**Likelihood:** High, since it can be triggered permissionlessly through a call to the bridge contract.

**Recommendation:** Fully constrain the global index in the contract. Check that bits over the 65th are zeroed out, and that the RER path is zeroed out if mainnet flag is set (finding "Mismatch in GlobalIndex parsing forces invalid proofs in AggKit").

**Polygon:** Fixed in agglayer-contracts PR 489.

**Spearbit:** Fix verified.

### 5.1.7 Duplicate certificate can corrupt Agglayer header store and DOS AggKit

**Severity:** High Risk

**Context:** mod.rs#L328-L336

**Summary:** A duplicate certificate can be submitted to the Agglayer after the original certificate is settled. This will overwrite the `Settled` certificate status with `Pending`, and the Agglayer will proceed without correcting the header status. The AggKit will see that the certificate is still `Pending` and will be unable to proceed.

**Finding Description:** When accepting certificates in the `send_certificate()` gRPC endpoint, a certificate is refused if there is an existing certificate for a specific `network_id` and `height` in the `pending_store` and it is not `InError`. This behavior can be seen in the `validate_pre_existing_certificate()` function at agglayer/crates/agglayer-rpc/src/lib.rs#L197-L294:

```
if let Some(certificate) = self
    .pending_store
    .get_certificate(certificate.network_id, certificate.height)?
{
    let pre_existing_certificate_id = certificate.hash();
    if let Some(CertificateHeader {
        status: CertificateStatus::InError { .. },
        settlement_tx_hash,
        ..
    }) = self
        .state
        .get_certificate_header(&pre_existing_certificate_id)?
    {
    // logic to replace cert header
    }
    // ....
}
```

After settling the certificate, it is removed from the `pending_store` at agglayer/crates/agglayer-storage/src/stores/per_epoch/mod.rs#L328-L336 This means that the `validate_pre_existing_certificate()` function will not see the pre-existing certificate and `send_certificate()` will begin accepting certificates again for the `network_id` at `height`.

If `send_certificate()` is called again with a duplicate certificate after the original certificate is settled but before a new certificate is sent, the cert will be inserted into the `pending_store` and the `state_store` again. See agglayer/crates/agglayer-rpc/src/lib.rs#L351-L358:

```
self.pending_store
    .insert_pending_certificate(certificate.network_id, certificate.height, &certificate)
    .inspect_err(|e| error!("Failed to insert certificate into pending store: {e}"))?;

// Insert the certificate header into the state store.
self.state
    .insert_certificate_header(&certificate, CertificateStatus::Pending)
    .inspect_err(|e| error!("Failed to insert certificate into state store: {e}"))?;
```

This will update the header to `CertificateStatus::Pending`. The duplicate certificate ends up silently failing during processing later in `make_progress()` at agglayer-certificate-orchestrator/src/network_task.rs#L209-L221 Since `next_expected_height` has been incremented from when the certificate was originally settled, the Agglayer will proceed on without attempting to reconcile the header status, and it will remain `Pending` forever.

When AggKit polls the Agglayer `GetCertificateHeader()` gRPC endpoint to see the certificate status at aggkit/aggsender/aggsender.go#L405, the Agglayer will respond that the certificate is `Pending`, even though it has

settled on moved on. This will make AggKit unable to continue sending certificates.

**Impact Explanation:** Medium - AggKit will be unable to produce any further certificates until the problem is identified and the offending header status is manually changed to `Settled` or `InError` in Agglayer. This would constitute a recoverable L2 chain DOS.

**Likelihood Explanation:** High - The attack scenario is permissionless, because the certificate required to trigger this must be a duplicate certificate that has the same certificate_id hash. All of the information required to build the certificate is made public except for the ECDSA signature.

In the case that the original certificate is an ECDSA certificate, there could be difficulty in obtaining the certificate signature, because the signature is never made public. In this case, the attacker can circumvent the signature entirely by faking that the certificate is non-ECDSA. This is possible because the `aggchain_data` field in the certificate is not included in the `certificate_id` hash, so a certificate that has the same `certificate_id` can have different `aggchain_data`. This fake `aggchain_data` could be made to not be the ECDSA type, so there will be no signature verification in `send_certificate()`. Since the certificate gets silently failed before getting processed in `make_progress()`, we do not need to worry about producing a valid `aggchain_data`.

**Proof of Concept:** This proof of concept is a regression test placed in agglayer/tests/integrations/tests/certificate_settlement/retries.rs and run with command `cargo nextest run --workspace -P integrations retries::regression_pushing_certificate_after_settling::case_1_type_0_ecdsa`. Further setup and instructions using the docs at `https://github.com/agglayer/agglayer?tab=readme-ov-file#running-the-tests`.

```rust
/// Validate that a certificate that has been proven and sent to L1 can't be
/// replaced in the certificate header store.
#[rstest]
#[tokio::test]
#[timeout(Duration::from_secs(180))]
#[case::type_0_ecdsa(crate::common::type_0_ecdsa_forest())]
async fn regression_pushing_certificate_after_settling(#[case] state: Forest) {
    let tmp_dir = TempDBDir::new();
    let scenario = FailScenario::setup();

    // L1 is a RAII guard
    let (_shutdown_shutdown, _l1, client) = setup_network(&tmp_dir.path, None, None).await;

    let withdrawals = vec![];

    let certificate = state.clone().apply_events(&[], &withdrawals);

    let first_certificate_id: CertificateId = client
        .request("interop_sendCertificate", rpc_params![certificate.clone()])
        .await
        .unwrap();

    // Send the first certificate. This should be settled.
    wait_for_settlement_or_error!(client, first_certificate_id).await;

    // Verify status is Settled.
    let first_header: CertificateHeader = client
        .request("interop_getCertificateHeader",rpc_params![first_certificate_id])
        .await
        .unwrap();
    assert!(matches!(first_header.status, CertificateStatus::Settled));

    // Send the second certificate, identical to the first.
    // @audit This currently does not error, but it should with fix.
    let second_certificate_id: CertificateId = client
        .request("interop_sendCertificate", rpc_params![certificate.clone()])
        .await
        .unwrap();
```

```
    assert_eq!(first_certificate_id, second_certificate_id);

    // Optional await sufficient time for cert to be processed.
    tokio::time::sleep(Duration::from_secs(10)).await;

    // Verify status is Settled.
    let second_header: CertificateHeader = client
        .request("interop_getCertificateHeader",rpc_params![second_certificate_id])
        .await
        .unwrap();

    // @audit This assert currently panics. The second_header.status becomes
    // CertificateStatus::Pending instead of CertificateStatus::Settled.
    assert!(matches!(second_header.status, CertificateStatus::Settled));

    scenario.teardown();
}
```

**Recommendation:** The `validate_pre_existing_certificate()` function should check for the non-`InError` certificate header in the `state_store` regardless of if the certificate exists in the `pending_store`.

**Polygon:** Fixed in PR 891.

**Spearbit:** Fix verified.

## 5.2 Medium Risk

### 5.2.1 Nullifier tree manipulation enables double-spending attacks

**Severity:** Medium Risk

**Context:** nullifier_tree.rs#L15-L33, main.rs#L9-L11

**Summary:** The pessimistic proof program implementation allows provers to manipulate `nullifier_tree.empty_hash_at_height` parameters, enabling double-spending attacks that fundamentally compromise the system's security guarantees. By providing malicious empty hash values, provers can reuse nullifiers that should be permanently consumed.

**Description:** The nullifier tree is designed to prevent double-spending by tracking used nullifiers in a Sparse Merkle Tree (SMT) structure. When a nullifier is consumed, it should be permanently recorded in the tree, making subsequent attempts to use the same nullifier detectable and rejectable.

However, the current implementation allows provers to supply their own `initial_state.nullifier_tree.empty_hash_at_height` parameter during the pessimistic proof generation. This input parameter defines what constitutes an "empty" position in the nullifier tree at each height level. By manipulating these values, a malicious prover can create inconsistent tree states where:

1. A nullifier appears to be successfully inserted into the tree (generating a valid proof).
2. The same nullifier is reused by referencing the manipulated empty hash values.
3. The pessimistic proof program treats the reused nullifier as valid, enabling double-spending.

The vulnerability stems from the trust in prover-supplied `empty_hash_at_height` parameter. The `empty_hash_at_height` array represents a fundamental tree structure property and should be computed/hardcoded inside of the pessimistic proof program rather than accepted from the provers' inputs.

**Impact Explanation:** High. This vulnerability completely undermines the nullifier core security guarantee of preventing double-spending at the ZK proof level.

**Likelihood Explanation:** Low. Compromised or malicious prover is required for this exploit. Provers have direct control over the `empty_hash_at_height` parameters and can manipulate them to enable the attack.

Moreover, this would require double spending an imported bridge exit from the chain. This could happen in case of a compromised chain (i.e. compromised aggkit).

**Proof of Concept:** The following proof of concept shows that the `empty_hash_at_height` parameter can be manipulated to make `NullifierTree::verify_and_update` succeed even if the nullifier key has already been used:

```
  diff --git a/crates/pessimistic-proof-core/src/nullifier_tree.rs
  ↪  b/crates/pessimistic-proof-core/src/nullifier_tree.rs
  index 81b8adc..5edb75a 100644
-  -- a/crates/pessimistic-proof-core/src/nullifier_tree.rs
+  ++ b/crates/pessimistic-proof-core/src/nullifier_tree.rs
  @@ -84,3 +84,131 @@ where
           Ok(())
       }
   }
+
+  #[cfg(test)]
+  mod tests {
+      use super::*;
+      use agglayer_primitives::keccak::Keccak256Hasher;
+      use unified_bridge::bridge_exit::NetworkId;
+
+      fn create_test_tree() -> NullifierTree<Keccak256Hasher> {
+          let mut empty_hash_at_height = [<Keccak256Hasher as
↪   agglayer_primitives::keccak::Hasher>::Digest::default(); NULLIFIER_TREE_DEPTH];
+          for height in 1..NULLIFIER_TREE_DEPTH {
+              empty_hash_at_height[height] = Keccak256Hasher::merge(
+                  &empty_hash_at_height[height - 1],
+                  &empty_hash_at_height[height - 1],
+              );
+          }
+          let root = Keccak256Hasher::merge(
+              &empty_hash_at_height[NULLIFIER_TREE_DEPTH - 1],
+              &empty_hash_at_height[NULLIFIER_TREE_DEPTH - 1],
+          );
+          NullifierTree {
+              root,
+              empty_hash_at_height,
+          }
+      }
+
+      fn create_test_key() -> NullifierKey {
+          NullifierKey {
+              network_id: NetworkId::from(0u32),
+              let_index: 0,
+          }
+      }
+
+      /// Helper function to create siblings for empty tree proof
+      fn create_empty_tree_siblings(tree: &NullifierTree<Keccak256Hasher>)
+          -> Vec<<Keccak256Hasher as agglayer_primitives::keccak::Hasher>::Digest> {
+
+          let mut siblings = Vec::new();
+          for i in 0..NULLIFIER_TREE_DEPTH {
+              siblings.push(tree.empty_hash_at_height[NULLIFIER_TREE_DEPTH - 1 - i]);
+          }
+          siblings
+      }
+
+      /// @POC: Helper function to create a tree with manipulated empty hash values
+      fn create_exploit_tree(original_tree: &NullifierTree<Keccak256Hasher>)
```

```
+          -> NullifierTree<Keccak256Hasher> {
+
+          let mut manipulated_empty_hashes = Vec::new();
+
+          // @POC: Try to manipulate the empty hash values to enable double-spend
+          // Here, we include hash(true) as an "empty" value which is unexpected
+          let true_value = <Keccak256Hasher as
↪  agglayer_primitives::keccak::Hasher>::Digest::from_bool(true);
+          manipulated_empty_hashes.push(true_value);
+
+          // @POC: Reuse the original empty hashes for the rest of the empty hashes
+          for i in 1..NULLIFIER_TREE_DEPTH {
+              manipulated_empty_hashes.push(original_tree.empty_hash_at_height[i]);
+          }
+
+          NullifierTree::<Keccak256Hasher> {
+              root: original_tree.root,
+              empty_hash_at_height: manipulated_empty_hashes.as_slice().try_into().unwrap(),
+          }
+      }
+
+
+      #[test]
+      fn test_double_spending_exploit() {
+          println!("=== Testing Nullifier Tree Double-Spend Exploit ===\n");
+
+          // Step 1: Setup initial state
+          let mut original_tree = create_test_tree();
+          let target_key = create_test_key();
+          let initial_root = original_tree.root;
+
+          println!("Initial tree root: 0x{}", hex::encode(initial_root.as_ref()));
+          println!("Target key - network_id: {}, let_index: {}\n",
+                   *target_key.network_id, target_key.let_index);
+
+          // Step 2: Create valid non-inclusion proof for empty tree
+          let valid_siblings = create_empty_tree_siblings(&original_tree);
+          let valid_path = NullifierPath {
+              siblings: valid_siblings
+          };
+
+          // Step 3: Perform legitimate nullifier insertion
+          println!("--- Legitimate Nullifier Insertion ---");
+          let legitimate_result = original_tree.verify_and_update(target_key, &valid_path);
+
+          match legitimate_result {
+              Ok(_) => {
+                  println!(" Legitimate insertion succeeded");
+                  println!("New tree root: 0x{}", hex::encode(original_tree.root.as_ref()));
+                  assert_ne!(original_tree.root, initial_root, "Root should change after insertion");
+              }
+              Err(e) => {
+                  panic!("Legitimate insertion failed: {:?}", e);
+              }
+          }
+
+          // Step 4: Attempt exploit - try to manipulate empty_hash_at_height
+          println!("\n--- Attempting Exploit ---");
+          println!("Trying to reuse the same key by manipulating empty hash values...");
+
+          let mut manipulated_tree = create_exploit_tree(&original_tree);
+          let exploit_siblings = create_empty_tree_siblings(&original_tree);
```

```
+          let exploit_path = NullifierPath {
+              siblings: exploit_siblings
+          };
+
+          // Step 5: Test the exploit
+          let exploit_result = manipulated_tree.verify_and_update(target_key, &exploit_path);
+
+          match exploit_result {
+              Ok(_) => {
+                  println!(" CRITICAL: Exploit succeeded! Double-spend possible!");
+                  println!("Exploit tree root: 0x{}", hex::encode(manipulated_tree.root.as_ref()));
+              }
+              Err(ProofError::InvalidNullifierPath) => {
+                  println!(" Exploit correctly prevented - InvalidNullifierPath error");
+                  println!("The nullifier tree successfully rejected the double-spend attempt");
+              }
+              Err(e) => {
+                  println!(" Exploit prevented with error: {:?}", e);
+              }
+          }
+      }
+  }
```

Then, start the test with the following command.

```
cargo test --package pessimistic-proof-core --lib --
↪  nullifier_tree::tests::test_double_spending_exploit --exact --show-output
```

The test results demonstrate successful exploitation:

```
---- nullifier_tree::tests::test_double_spending_exploit stdout ----
=== Testing Nullifier Tree Double-Spend Exploit ===

Initial tree root: 0xe2c3ed4052eeb1d60514b4c38ece8d73a27f37fa5b36dcbf338e70de95798caa
Target key - network_id: 0, let_index: 0

--- Legitimate Nullifier Insertion ---
 Legitimate insertion succeeded
New tree root: 0x56db1d35c05310a3dd9ee68c6271e822543be183d78d08727b471e7d989471ee

--- Attempting Exploit ---
Trying to reuse the same key by manipulating empty hash values...
 CRITICAL: Exploit succeeded! Double-spend possible!
Exploit tree root: 0x56db1d35c05310a3dd9ee68c6271e822543be183d78d08727b471e7d989471ee
```

**Recommendation:** Consider implementing deterministic computation of `empty_hash_at_height` values within the pessimistic proof program itself rather than accepting them from provers. This will ensure the correctness of these values. Only the `nullifier_tree.root` should come from the provers.

```
impl<H: Hasher> NullifierTree<H> {
    // Compute empty hash values deterministically
    fn compute_empty_hashes() -> [H::Digest; NULLIFIER_TREE_DEPTH] {
        let mut empty_hash_at_height = [H::Digest::default(); NULLIFIER_TREE_DEPTH];
        for height in 1..NULLIFIER_TREE_DEPTH {
            empty_hash_at_height[height] = H::merge(
                &empty_hash_at_height[height - 1],
                &empty_hash_at_height[height - 1],
            );
        }
        empty_hash_at_height
    }
}
```

According to the pessimistic proof benchmark, keccak computations already represent 75% of the computation in the zkVM. Considering this, hardcoding the `empty_hash_at_height` values is recommended to avoid redundant computations.

**Polygon:** Fixed in PR73 of interop repository.

**Spearbit:** Fixed. The `SmtNonInclusionProof::verify_and_update` function does not expect the `empty_hash_at_height` anymore. Instead, the `EMPTY_HASH_ARRAY_AT_193` constant holds the 193 hashes that corresponds to the empty hashes at each level (from 0 to 192).

> **Disclaimer:** This issue will be fixed once the new version of interop that includes this fix is used by the `pessimistic-proof-core`.

#### 5.2.2 V2 commitments don't include the height

**Severity:** Medium Risk

**Context:** lib.rs#L327-L353

**Description:** The PP+ECDSA (V0.2.0) chains are prone to a denial of service attack vector at the Agglayer level. An attacker can submit a certificate in which the height has been modified such that no subsequent certificates are accepted. The height of the certificate can be manipulated without modifying the signature attached to the certificate. This is because V2 commitments do not include the height (see `SignatureCommitmentValues.commitment`).

```
    pub fn signer(&self) -> Result<Option<Address>, SignatureError> {
        match self.aggchain_data {
            AggchainData::ECDSA { signature } => {
                // retrieve signer
                let version = CommitmentVersion::V2;
                let combined_hash = SignatureCommitmentValues::from(self).commitment(version);

                signature
                    .recover_address_from_prehash(&B256::new(combined_hash.0))
                    .map(Some)
            }
            _ => Ok(None),
        }
    }
}

impl SignatureCommitmentValues {
    /// Returns the expected signed commitment for the provided version.
    #[inline]
    pub fn commitment(&self, version: CommitmentVersion) -> Digest {
        let imported_bridge_exit_commitment = self.commit_imported_bridge_exits.commitment(version);
        match version {
            CommitmentVersion::V2 => keccak256_combine([
                self.new_local_exit_root.as_slice(),
                imported_bridge_exit_commitment.as_slice(),
            ]),
            CommitmentVersion::V3 => keccak256_combine([
                self.new_local_exit_root.as_slice(),
                imported_bridge_exit_commitment.as_slice(),
                self.height.to_le_bytes().as_slice(),
            ]),
        }
    }
}
```

*Note: V3 commitments are not impacted.*

**Impact Explanation:** Medium: The Agglayer does not process certificate for the chain anymore because it has a certificate with a height greater than the one being submitted. This leads to a denial of service for the chain. `insert_pending_certificate` only accepts certificates with a height greater than the height of the latest certificate.

```rust
fn insert_pending_certificate(
    &self,
    network_id: NetworkId,
    height: Height,
    certificate: &Certificate,
) -> Result<(), Error> {
    if let Some((_id, latest_height)) =
        self.get_latest_pending_certificate_for_network(&network_id)?
    {
        if latest_height > height { // @audit error if the new is less than the latest
            // TODO: This is technically not Candidate error,
            return Err(Error::CertificateCandidateError(
                crate::error::CertificateCandidateError::UnexpectedHeight(
                    network_id,
                    height,
                    latest_height,
                ),
            ));
        }
    }
```

**Likelihood Explanation: Medium**: Executing such attack requires accessing to a certificate with a valid signature.

**Recommendation:** This issue is fixed for V3 commitments. V2 commitments are technical debt.

**Polygon:** Acknowledged. We acknowledge that V2 may cause issues (cf. agglayer/protocol-research#120), which motivated the definition of the V3 commitment, towards which we are transitioning.

**Spearbit:** Acknowledged.

## 5.3 Low Risk

### 5.3.1 AggKit does not properly handle removed GER to comply with Aggchain proof

**Severity:** Low Risk

**Context:** evm.go#L110-L117, GlobalExitRootManagerL2SovereignChain.sol#L23-L25, lib.rs#L249-L258, mod.rs#L180-L198

**Description:** The `EVMChainGERReader.GetInjectedGERsForRange` function is called from the AggSender component to collect the inserted GERs in the L2 block range being proven. These GERs are used as inputs to the AggKit prover to generate the Aggchain proof. Then, the Aggchain proof program will ensure that all the inserted GERs were correctly by computing and verifying the `insertedGERHashChain` parameter.

However, the current implementation of `GetInjectedGERsForRange` unexpectedly removes the removed GERs from the inserted GERs. This will lead to computing an incorrect `insertedGERHashChain` hash chain when a GER is inserted and removed in the same block range being proven.

**Impact:** Medium. This issue leads to an Agglayer denial of service for the L2 as the Aggchain proof program will fail to verify the inserted GERs hash chain.

**Likelihood:** Low. This issue requires inserting and removing a GER in the same L2 block range.

Code snippets: The AggKit removes the removed GERs from the inserted ones. This is incorrect as they should be handled in separate processing. This is because when a GER is removed, it can't be removed from the inserted GER hash chain.

```
// GetInjectedGERsForRange returns the injected GlobalExitRoots for the given block range
func (e *EVMChainGERReader) GetInjectedGERsForRange(ctx context.Context,
    fromBlock, toBlock uint64) (map[common.Hash]InjectedGER, error) {
    // ...

    for insertIter.Next() {
        if insertIter.Error() != nil {
            return nil, insertIter.Error()
        }

        ger := insertIter.Event.NewGlobalExitRoot
        injectedGERs[ger] = InjectedGER{ // @POC: add GER to injected GERs
            BlockNumber:    insertIter.Event.Raw.BlockNumber,
            BlockIndex:     insertIter.Event.Raw.Index,
            GlobalExitRoot: ger,
        }
    }

    // ...

    for removalIter.Next() {
        if removalIter.Error() != nil {
            return nil, removalIter.Error()
        }

        ger := removalIter.Event.RemovedGlobalExitRoot
        delete(injectedGERs, ger) // @POC: if the GER has been removed, delete it from the injected
        ↪  GERs. This breaks the hash chain
    }

    // ...
}
```

The `AggchainProofBuilder::retrieve_chain_data` function takes this inserted GERs input and include it as the `bridge_witness.raw_inserted_gers`. Then, the Aggchain proof program is not able to prove the injected GERs because the removed ones are missing from the inputs.

```
/// Verify the previous and new hash chains and their reconstructions.
fn verify_ger_hash_chains(&self) -> Result<(), BridgeConstraintsError> {
    // Verify the hash chain on inserted GER
    {
        let hash_chain_type = HashChainType::InsertedGER;
        let (prev_hash_chain, new_hash_chain): (Digest, Digest) = self
            .fetch_hash_chains(
                hash_chain_type,
                self.ger_addr,
                GlobalExitRootManagerL2SovereignChain::insertedGERHashChainCall {},
            )
            .map(|(prev, new)| (prev.hashChain.0.into(), new.hashChain.0.into()))?;

        self.validate_hash_chain( // @POC: this will fail because inserted GER that have been removed
        ↪  are missing
            &self.bridge_witness.raw_inserted_gers,
            prev_hash_chain,
            new_hash_chain,
            hash_chain_type,
        )?;
    }
```

The `GlobalExitRootManagerL2SovereignChain` indicates that the AggchainFEP chains should not allow removing

GER. However, it does not mention the AggchainECDSA chains should also not support it.

```
// globalExitRootRemover address
// In case of initializing a chain with Full execution proofs, this address should be set to zero,
↪  otherwise, some malicious sequencer could insert invalid global exit roots, claim, go back and the
↪  execution would be correctly proved.
address public globalExitRootRemover;
```

**Recommendation:** The AggKit node should keep the inserted GER to avoid breaking the hash chain. This can be done by handling separately the injected GERs and the removed GERs. Moreover, the Aggchain proof program should completely deny the removal of GERs if these are not expected. If such change is applied, the AggKit should be modified to satisfy this.

**Polygon:** Acknowledged. Fix in progress on AggKit code in PR 883.

**Spearbit:** Acknowledged.

#### 5.3.2 `queryBlockRange` SQL request lacks ordering

**Severity:** Low Risk

**Context:** processor.go#L163-L205

**Description:**

1. The `queryBlockRange` function gets data from the SQL database in a given block range. However, the SQL request does not ensure that the data is ordered ascendingly by block number and block position.
2. The `queryBlockRange` function retrieves data from the SQL database within a specified block range. However, the underlying SQL query lacks an explicit ordering mechanism, which means the returned data is not guaranteed to be sorted in ascending order by block number and block position.

**Recommendation:**

1. Consider adding the `ORDER BY` parameter to ensure the data is in correct order.
2. Add an `ORDER BY` clause to the SQL query to explicitly sort results by block number and block position in ascending order.

   This will ensure data consistency and prevent issues arising from unpredictable result ordering that may vary between database implementations or query executions.

**Polygon:** Fixed in aggkit PR 658.

**Spearbit:** Fixed. Additional `ORDER BY` has been added in the SQL request.

#### 5.3.3 AggchainECDSA `newStateRoot` is not constrained against ZK public inputs

**Severity:** Low Risk

**Context:** AggchainECDSA.sol#L230-L234

**Description:** The `AggchainECDSA` contract does not constrain the `newStateRoot` value in `aggchainData` against the ZK proof public inputs in `getAggchainHash()`.

```
function getAggchainHash(
    bytes memory aggchainData
) external view returns (bytes32) {
    if (aggchainData.length != 32 * 2) {
        revert InvalidAggchainDataLength();
    }

    // The second param is the new state root used at onVerifyPessimistic callback but now only
    ↪  aggchainVKeySelector is required
    (bytes4 aggchainVKeySelector, ) = abi.decode( // @audit: the 32 bytes of new state are ignored
        aggchainData,
        (bytes4, bytes32)
    );
```

However, this `newStateRoot` value is emitted in the `onVerifyPessimistic()` function. This value can't be trusted as it has not been verified.

```
function onVerifyPessimistic(
    bytes calldata aggchainData
) external onlyRollupManager {
    if (aggchainData.length != 32 * 2) {
        revert InvalidAggchainDataLength();
    }

    (, bytes32 newStateRoot) = abi.decode(aggchainData, (bytes4, bytes32));

    // Emit event
    emit OnVerifyPessimisticECDSA(newStateRoot); // @POC: emit newStateRoot
}
```

**Recommendation:** `newStateRoot` should be decoded in `getAggchainHash` and constrained against the ZK public inputs to ensure its correctness.

**Polygon:** Fixed. The `newStateRoot` emitted in the event is for informational purposes. There is no productive `AggchainECDSA` and this kind of aggchains will be deprecated soon for their low level of security in terms of consensus.

A comment has been added in agglayer-contracts PR 501.

**Spearbit:** Fixed. The additional comment indicates that this parameter should not be trusted. Moreover, `AggchainECDSA` is likely to be deprecated.

#### 5.3.4 Hardcoded `removed_gers` vector may lead to Aggchain proof generation denial of service

**Severity:** Low Risk

**Context:** lib.rs#L342, mod.rs#L200-L217

**Description:** The `BridgeWitness.removed_gers` is hardcoded to an empty vector in the `aggchain-proof-builder` component.

```
bridge_witness: BridgeWitness {
    inserted_gers,
    bridge_exits_claimed,
    global_indices_unset: vec![],
    raw_inserted_gers: inserted_gers_hash_chain,
    removed_gers: vec![], // @audit this may lead to ZK proof generation DoS
    prev_l2_block_sketch,
    new_l2_block_sketch,
},
```

However, `GlobalExitRootManagerL2SovereignChain.removeGlobalExitRoots()` can still be used on the L2. When this function is called to remove a GER, all subsequent Aggchain proof generation attempt will fail because the removed GERs on L2 don't match `BridgeWitness.removed_gers`.

**Recommendation:** Consider retrieving the removed GERs in the AggKit and adding them to the `GenerateAggchainProofRequest` such that the Aggchain provers can prove these removed GERs. Another way to fix this is to remove the `GlobalExitRootManagerL2SovereignChain.removeGlobalExitRoots()` functionality.

**Polygon:** Acknowledged, not yet fixed. This fix will be done on v0.4.0.

**Spearbit:** Acknowledged.

#### 5.3.5 Hardcoded `global_indices_unset` vector may lead to Aggchain proof generation denial of service

**Severity:** Low Risk

**Context:** lib.rs#L340, mod.rs#L248-L271

**Description:** The `BridgeWitness.global_indices_unset` is hardcoded to an empty vector in the `aggchain-proof-builder` component.

```
bridge_witness: BridgeWitness {
    inserted_gers,
    bridge_exits_claimed,
    global_indices_unset: vec![], // @audit this may lead to ZK proof generation DoS
    raw_inserted_gers: inserted_gers_hash_chain,
    removed_gers: vec![],
    prev_l2_block_sketch,
    new_l2_block_sketch,
},
```

However, `BridgeL2SovereignChain.unsetMultipleClaims()` can still be used on the L2. When this function is called to remove a claim from the claimed bitmap, all subsequent Aggchain proof generation attempt will fail because the unset claims (and `unsetGlobalIndexHashChain`) on L2 don't match `BridgeWitness.global_indices_unset`.

**Recommendation:** Consider retrieving the unset claims in the AggKit and adding them to the `GenerateAggchainProofRequest` such that the Aggchain provers can prove these global indices as not claimed. Another way to fix this is to remove the `BridgeL2SovereignChain.unsetMultipleClaims()` functionality.

**Polygon:** Acknowledged, fix in progress. This will be done on v0.4.0.

**Spearbit:** Acknowledged.

#### 5.3.6 `unsetMultipleClaims` functionality is not compatible with the pessimistic proof

**Severity:** Low Risk

**Context:** BridgeL2SovereignChain.sol#L608-L610

**Description:** The `BridgeL2SovereignChain.unsetMultipleClaims()` function allows to remove a given claim from the `claimedBitMap` bitmap. This function allows to claim a message or asset multiple times at the chain level. However, the pessimistic proof includes the nullifier tree which does not support double spending a claim. As a result, double spending of a claim on the L2 through the use of `BridgeL2SovereignChain.unsetMultipleClaims()` leads to the denial of service of the pessimistic proof generation.

The `_unsetClaimedBitmap()` function is called from `unsetMultipleClaims()` to remove a claim from the `claimedBitMap` variable.

```
function _unsetClaimedBitmap(
    uint32 leafIndex,
    uint32 sourceBridgeNetwork
) private {
    uint256 globalIndex = uint256(leafIndex) +
        uint256(sourceBridgeNetwork) *
        _MAX_LEAFS_PER_NETWORK;
    (uint256 wordPos, uint256 bitPos) = _bitmapPositions(globalIndex);
    uint256 mask = 1 << bitPos;
    uint256 flipped = claimedBitMap[wordPos] ^= mask; // @audit mark as not claimed
    if (flipped & mask != 0) {
        revert ClaimNotSet();
    }
}
```

This allows claiming multiple times a single claim. However, the Agglayer will not be able to prove the Nullifier Sparse Merkle Tree non-inclusion for this specific claim as it could be spent multiple times.

```
pub fn apply_certificate(
    &mut self,
    certificate: &Certificate,
    signer: Address,
    l1_info_root: Digest,
    prev_pp_root: PessimisticRootInput,
    aggchain_vkey: Option<Vkey>,
) -> Result<MultiBatchHeader<Keccak256Hasher>, Error> {
    // ...
    let imported_bridge_exits: Vec<(ImportedBridgeExit, NullifierPath<Keccak256Hasher>)> =
        certificate
            .imported_bridge_exits
            .iter()
            .map(|exit| {
                // ...
                let nullifier_path = self
                    .nullifier_tree
                    .get_non_inclusion_proof(nullifier_key) // @audit will error with
                    ↪  `SmtError::KeyPresent`
                    .map_err(nullifier_error)?;
```

This breaks the generation of the pessimistic proof.

**Recommendation:** The pessimistic proof is specifically built to avoid double spending claims. To keep this core feature of the Agglayer, the `BridgeL2SovereignChain.unsetMultipleClaims()` functionality should be removed.

**Polygon:** Acknowledged.

**Spearbit:** Acknowledged. This `unsetMultipleClaims` function is meant to be used when (incorrect) GER are removed to not lock the claim index. Double spending will not be recorded at the PP level. It is important to note that in case of an incorrect GER being inserted and then removed, the ZK proof generation will fail as the accumulators (hash chains) will not track correctly the events.

### 5.3.7 Certificates removed from the Agglayer will remain as "pending" leading the Aggsender to skip all subsequent epochs

**Severity:** Low Risk

**Context:** aggsender.go#L254-L267, aggsender.go#L394-L430, node_state_service.rs#L60-L78, lib.rs#L176-L186

**Description:** The Agglayer node has a `/admin/removePendingCertificate` JSON RPC endpoint that allows the administrator to remove a given certificate from the pending store. However, the certificate header status in the

state store is not updated when a certificate is removed from the pending store. This can lead the Aggsender to skip subsequent epochs because the certificate status is still marked as "pending" on the Agglayer.

- `Agglayer`: When the `/admin/removePendingCertificate` JSON RPC endpoint is used, the certificate is removed only from the pending store.

```rust
#[instrument(skip(self), level = "debug")]
async fn remove_pending_certificate(
    // ...
) -> RpcResult<()> {
    // ...

    self.pending_store
        .remove_pending_certificate(network_id, height) // @audit remove from the pending store
        .map_err(|error| {
            error!("Failed to remove pending certificate: {}", error);
            Error::internal("Unable to remove pending certificate")
        })?;

    // ...

    Ok(())
}
```

  However, the `GetCertificateHeader` gRPC endpoint used by the Aggsender will read the certificate header from the state store, and not the pending store.

```rust
pub fn fetch_certificate_header(
    &self,
    certificate_id: CertificateId,
) -> Result<CertificateHeader, CertificateRetrievalError> {
    self.state
        .get_certificate_header(&certificate_id) // @audit get the certificate
        .inspect_err(|err| error!("Failed to get certificate header: {err}"))?
        .ok_or(CertificateRetrievalError::NotFound { certificate_id })
}
```

  When a certificate is removed from the pending store, its certificate header status will never be modified. A pending certificate will remain in the pending status.

- `Aggsender`: For each epoch, the `Aggsender` checks if there are pending certificates.

```
func (a *AggSender) checkPendingCertificatesStatus(ctx context.Context) checkCertResult {
    // ...

    a.log.Debugf("checkPendingCertificatesStatus num of pendingCertificates: %d",
    ↪  len(pendingCertificates))
    thereArePendingCerts := false
    appearsNewInErrorCert := false
    for _, certificateLocal := range pendingCertificates {
        certificateHeader, err := a.aggLayerClient.GetCertificateHeader(ctx,
        ↪  certificateLocal.CertificateID) // @audit get the header from the state store
        // ...

        if !certificateLocal.IsClosed() {
            a.log.Infof("certificate %s is still pending, elapsed time:%s ",
                certificateHeader.ID(), certificateLocal.ElapsedTimeSinceCreation())
            thereArePendingCerts = true // @audit there are pending certificates if one is not
            ↪  closed (i.e. is not settled or in error)
        }
    }
    return checkCertResult{existPendingCerts: thereArePendingCerts, existNewInErrorCert:
    ↪  appearsNewInErrorCert}
```

The `Aggsender` skips the epoch when there is a pending certificate.

```
func (a *AggSender) sendCertificates(ctx context.Context, returnAfterNIterations int) {
    // ...
    for {
        // ...
        case epoch := <-chEpoch:
            iteration++
            a.log.Infof("Epoch received: %s", epoch.String())
            checkResult := a.checkPendingCertificatesStatus(ctx) // @audit get certificate
            ↪  status
            if !checkResult.existPendingCerts {
                _, err := a.sendCertificate(ctx)
                a.status.SetLastError(err)
                if err != nil {
                    a.log.Error(err)
                }
            } else { // @audit skip the epoch if not settled or in error
                log.Infof("Skipping epoch %s because there are pending certificates",
                    epoch.String())
            }
```

**Recommendation:** In the Agglayer `removePendingCertificate` JSON RPC endpoint, consider setting the certificate status to `InError` in the state store if the certificate has not been settled and is removed from the pending store.

**Polygon:** Fixed in agglayer PR 954.

**Spearbit:** Fixed. The certificate status is now changed to `CertificateStatusError::InternalError`.

### 5.3.8 Inconsistent states are reachable in Agglayer due to specific failures

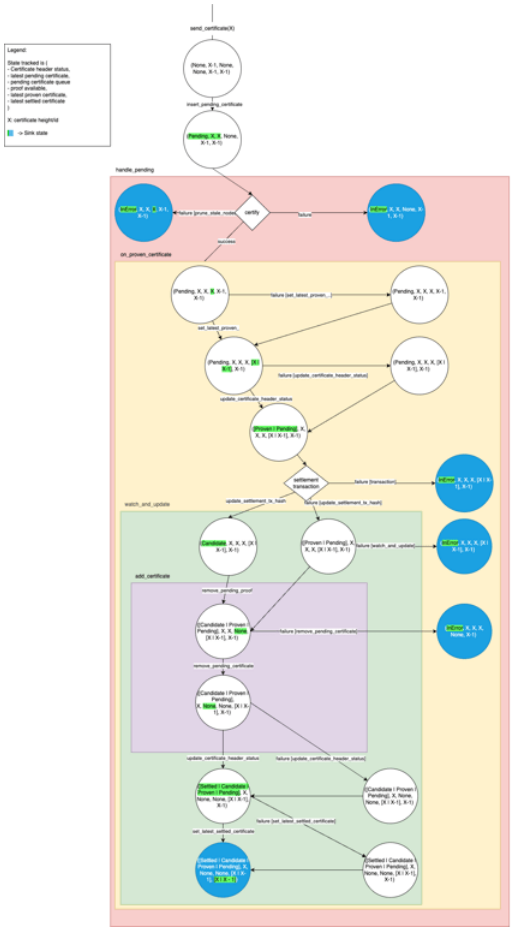
**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** Agglayer maintains a state with regard to the processing of a certificate, which can be listed as below:

```
- Certificate header status (key: CertificateHeaderColumn)
- latest pending certificate (key: LatestPendingCertificatePerNetworkColumn)
- pending certificate queue (key: PendingQueueColumn)
- proof available (key: ProofPerCertificateColumn)
- latest proven certificate (key: LatestProvenCertificatePerNetworkColumn)
- latest settled certificate (key: LatestSettledCertificatePerNetworkColumn)
```

Here's a state diagram showcasing the transitions and reachable states given failure points:



> For simplicity, we only consider `handle_pending` function since it can handle the processing of an inserted certificate all the way to settlement. Other processing functions, such as `handle_candidate`, will only try to pick the process where it stopped.

As we can see, the resulting state at the end of the processing pipeline (`handle_pending`) can be represented in a compact way as below:

| Certificate header status | latest pending certificate | pending certificate queue | proof available | latest proven certificate | latest settled certificate |
|---|---|---|---|---|---|
| `([Settled \ Candidate \ Proven \ Pending]` | X | None | None | `[X \ X - 1]` | `[X \ X - 1])` |

For example, if only the latest `update_certificate_header_status` (to set status to `Settled`) fails, the agglayer state as fetched from the aggkit for initialisation is inconsistent:

- aggsender/aggsender_initial_state.go#L198-L199:

```
if i.PendingCert.Height == i.SettledCert.Height &&
    !i.SettledCert.Status.IsInError() {
```

Similarly, if only the latest `set_latest_settled_certificate_for_network` call fails, the agglayer upon restart will be underestimating `next_expected_height` by one, and will not accept certificates sent by the corresponding aggkit:

- agglayer-certificate-orchestrator/src/network_task.rs#L118-L136:

```
// Start from the latest settled certificate to define the next expected height
let latest_settled = self
    .state_store
    .get_latest_settled_certificate_per_network(&self.network_id)?
    .map(|(_network_id, settled)| settled);

let mut next_expected_height =
    if let Some(SettledCertificate(_, current_height, epoch, _)) = latest_settled {
        debug!("Current network height is {}", current_height);
        if epoch == current_epoch {
            debug!("Already settled for the epoch {current_epoch}");
            self.at_capacity_for_epoch = true;
        }

        current_height + 1
    } else {
        debug!("Network never settled any certificate");
        0
    };
```

**Impact:** This impacts the Aggkit for the given network when either the Aggkit or Agglayer restarts and has to reconstruct local state from storage. The impact is DOS of the given network until manual intervention.

**Likelihood:** The likelihood is very low, since it requires very specific failure points: `set_latest_settled_certificate_for_network` or `update_certificate_header_status`, during the settlement, combined with a restart of Agglayer or Aggkit right after the failure.

**Recommendation:** Consider adding a routine to recover these state inconsistencies, for example, in `make_progress`:

- Systematically align the highest settled certificate header in `CertificateHeaderColumn` to `LatestSettledCertificatePerNetworkColumn` using `set_latest_settled_certificate_for_network`.
- Systematically update header status in `CertificateHeaderColumn` to align with `get_latest_settled_certificate_for_network`.

**Polygon:** Acknowledged, this will be fixed on the refactoring of the storage. No target version yet.

**Spearbit:** Acknowledged.

### 5.3.9 Aggregation prover can corrupt certificate metadata by using an empty range proof

**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** Aggkit prover and Aggkit are designed to handle partial FEP proofs; The aggregation prover can set the `end_block` for the aggregation proof to be lower than the "requested" `end_block`. Here we will analyze the edge case where the aggregation prover provides a proof verifying `end_block == last_proven_block` (empty aggregation proof). In most cases, `last_proven_block == from_block - 1`:

- flows/flow_aggchain_prover.go#L383-L391:

```
func (a *AggchainProverFlow) getLastProvenBlock(fromBlock uint64) uint64 {
    if fromBlock == 0 {
        // if this is the first certificate, we need to start from the starting L2 block
        // that we got from the sovereign rollup
        return a.startL2Block
    }

    return fromBlock - 1
}
```

Which means that the earliest `end_block` for a valid aggchain-proof would be `buildParams.fromBlock - 1`. As a result, the `ToBlock` of the certificate build params is reset to `buildParams.fromBlock - 1`:

- flows/flow_aggchain_prover.go#L208:

```
return adjustBlockRange(buildParams, buildParams.ToBlock, aggchainProof.EndBlock)
```

> Please note that `allowEmptyCert` should be true here, otherwise the empty certificate is not built/sent.

This, however, leads to a correct empty certificate to be built; The only difference is that the offset computed for metadata has an underflow:

- aggsender/flows/flow_base.go#L179-L183:

```
meta := types.NewCertificateMetadata(
    certParams.FromBlock,
    //@audit offset underflow because certParams.ToBlock == certParams.FromBlock - 1
    uint32(certParams.ToBlock-certParams.FromBlock),
    certParams.CreatedAt,
)
```

**Impact:** If such an empty certificate is sent, the Aggkit is not able to recover during restart due to corrupted metadata:

- aggkit/aggsender/aggsender.go#L523-L529:

```
func NewCertificateInfoFromAgglayerCertHeader(c *agglayertypes.CertificateHeader)
↪  *types.CertificateInfo {
    if c == nil {
        return nil
    }
    now := uint32(time.Now().UTC().Unix())
    meta := types.NewCertificateMetadataFromHash(c.Metadata)
    //@audit here meta.Offset is math.MaxUint32
    toBlock := meta.FromBlock + uint64(meta.Offset)
```

**Recommendation:** Protect against the edge case where the aggregation prover resets `EndBlock` to `LastProvenBlock`:

```
+ if aggchainProof.EndBlock == lastProvenBlock {
+     return nil, fmt.Err("Invalid EndBlock, is before FromBlock")
+ }
```

**Polygon:** Acknowledged, fix in progress. This will be done on v0.7 of AggKit.

**Spearbit:** Acknowledged.

### 5.3.10 Incorrect name for table "global_exit_root" in database migration

**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** Aggkit uses a declarative database migration system to manage different versions of the database model. We notice an inconsistency between the "up" and "down" directions for the `imported_global_exit_root` table in `aggkit/lastgersync/migrations/lastgersync0001.sql`:

- lastgersync/migrations/lastgersync0001.sql#L1-L14:

```
-- +migrate Down
DROP TABLE IF EXISTS block;
//@audit should be imported_global_exit_root
DROP TABLE IF EXISTS global_exit_root;

-- +migrate Up
CREATE TABLE block (
    num   BIGINT PRIMARY KEY
);

CREATE TABLE imported_global_exit_root (
    block_num           INTEGER PRIMARY KEY REFERENCES block(num) ON DELETE CASCADE,
    global_exit_root    VARCHAR NOT NULL,
    l1_info_tree_index  INTEGER NOT NULL
);
```

**Impact:** Reverting to a previous version of the database would not remove the `imported_global_exit_root`. The impact is very limited.

**Recommendation:** lastgersync/migrations/lastgersync0001.sql#L1-L14:

```
  -- +migrate Down
  DROP TABLE IF EXISTS block;
- DROP TABLE IF EXISTS global_exit_root;
+ DROP TABLE IF EXISTS imported_global_exit_root;
```

**Polygon:** Fixed in aggkit PR 611.

**Spearbit:** Fix verified.

#### 5.3.11 Certificate without bridge exits after reduction may be sent even if `!allowEmptyCert`

**Severity:** Low Risk

**Context:** flow_base.go#L133

**Description:** There is a reduction step during build params to ensure a certificate does not exceed a configured size limit (`cfg.MaxCertSize`). After this phase, if the certificate ends up not containing any bridge exits, the certificate params are still forwarded without error for certificate building; As a result, an empty certificate will be sent by aggkit to the agglayer:

- aggkit/aggsender/flows/flow_base.go#L130-L134:

```
if currentCert.NumberOfBridges() == 0 && !allowEmptyCert {
    f.log.Warnf("Minimum certificate size reached. Estimated size: %d > max size: %d",
        currentCert.EstimatedSize(), f.maxCertSize)
    //@audit currentCert is returned without error, even if !allowEmptyCert. (nil, nil) should be
    ↪  returned instead
    return currentCert, nil
}
```

> Please note that the configuration `MaxCertSize` was introduced to work with limitations intrinsic to json-rpc, and with the migration to gRPC, may not be relevant anymore.

**Impact:** Impact is limited, since the empty certificate is still correctly handled by agglayer and contracts, not emitting a new event for insertion of the unchanged GER.

**Recommendation:** Please consider returning empty build params instead:

- aggkit/aggsender/flows/flow_base.go#L130-L134:

```
   if currentCert.NumberOfBridges() == 0 && !allowEmptyCert {
       f.log.Warnf("Minimum certificate size reached. Estimated size: %d > max size: %d",
           currentCert.EstimatedSize(), f.maxCertSize)
-      return currentCert, nil
+      return nil, nil
   }
```

**Polygon:** Fixed in PR 619.

**Spearbit:** Fix verified.

#### 5.3.12 `CommitmentVersion::V3` for ECDSA proof is not handled in all code paths in agglayer

**Severity:** Low Risk

**Context:** lib.rs#L445-L447

**Description:** `CommitmentVersion::V3` introduced for consensus type 1, is not fully compatible with ECDSA proofs on agglayer side.

> Please note that the docs correctly specify that no aggkit is ready to use this type of consensus:
> `v0.3.0-ECDSA ALGateway AggchainECDSA PP # SP1 not supported (aggkit not built).`
> Here we are reporting the non-readiness of Agglayer to handle this consensus type as well.

While it is handled in some code paths, such as below with the migration from CommitmentVersion::V2 to CommitmentVersion::V3 in the ECDSA proof case:

- pessimistic-proof-core/src/proof.rs#L291-L300:

```
//@audit Transition from AggchainData::ECDSA to AggchainData::ECDSA
match (base_pp_root_version, target_pp_root_version) {
    // From V2 to V2: OK
    (CommitmentVersion::V2, CommitmentVersion::V2) => {}
    // From V3 to V3: OK
    (CommitmentVersion::V3, CommitmentVersion::V3) => {}
    // From V2 to V3: OK (migration)
    //@audit enables AggchainData::ECDSA with CommitmentVersion::V3
    (CommitmentVersion::V2, CommitmentVersion::V3) => {}
    // Inconsistent signed payload.
    _ => return Err(ProofError::InconsistentSignedPayload),
}
```

We can see that in some cases, the usage of AggchainData::ECDSA proofs with CommitmentVersion::V3 would fail:

1. `aggchain_hash` can be only evaluated for AggchainData::ECDSA in v0.2.0:

   pessimistic-proof-core/src/aggchain_proof.rs#L47-L67:

```rust
pub fn aggchain_hash(&self) -> Digest {
    match &self {
        AggchainData::ECDSA { signer, .. } => keccak256_combine([
            &(ConsensusType::ECDSA as u32).to_be_bytes(),
            signer.as_slice(),
        ]),
        //@audit need to handle aggchain_hash generation with format for v0.3.0:
        /*
            keccak256_combine([
                &(ConsensusType::Generic as u32).to_be_bytes(),
                aggchain_vkey_hash.as_slice(),
                keccak256(signer).as_slice(),
            ])
        */
        AggchainData::Generic {
            aggchain_params,
            aggchain_vkey,
        } => {
            let mut aggchain_vkey_hash = [0u8; 32];
            BigEndian::write_u32_into(aggchain_vkey, &mut aggchain_vkey_hash);

            keccak256_combine([
                &(ConsensusType::Generic as u32).to_be_bytes(),
                aggchain_vkey_hash.as_slice(),
                aggchain_params.as_slice(),
            ])
        }
    }
}
```

2. `signer()` is only evaluated using `CommitmentVersion::V2`:

   agglayer-types/src/lib.rs#L442-L456:

```rust
pub fn signer(&self) -> Result<Option<Address>, SignatureError> {
    match self.aggchain_data {
        AggchainData::ECDSA { signature } => {
            // retrieve signer
            //@audit Only CommitmentVersion::V2 is used. If CommitmentVersion::V3 is provided,
            ↪  signature verification would fail
            let version = CommitmentVersion::V2;
            let combined_hash = SignatureCommitmentValues::from(self).commitment(version);

            signature
                .recover_address_from_prehash(&B256::new(combined_hash.0))
                .map(Some)
        }
        _ => Ok(None),
    }
}
```

**Recommendation:** Adapt existing code to handle AggchainData::ECDSA with CommitmentVersion::V3 in all paths of the agglayer code.

**Polygon:** Fixed in PR 920.

**Spearbit:** Fix verified.

### 5.3.13 Tx.origin for checking top-level call is not reliable post EIP-7702

**Severity:** Low Risk

**Context:** PolygonRollupBaseEtrog.sol#L562-L565

**Description:** The `forceBatch` function used for the zkEVM state transition type determines if the call is top-level or internal based on whether `msg.sender == tx.origin`. This assumption will be incorrect after EIP-7702 is introduced:

- PolygonZkEVM.sol#L1050-L1062:

```
if (msg.sender == tx.origin) {
    // Getting the calldata from an EOA is easy so no need to put the `transactions` in the
    ↪  event
    emit ForceBatch(lastForceBatch, lastGlobalExitRoot, msg.sender, "");
} else {
    // Getting internal transaction calldata is complicated (because it requires an archive
    ↪  node)
    // Therefore it's worth it to put the `transactions` in the event, which is easy to query
    emit ForceBatch(
        lastForceBatch,
        lastGlobalExitRoot,
        msg.sender,
        transactions
    );
}
```

**Recommendation:** Even though the impact is limited, consider removing the criteria to prepare for EIP-7702:

- PolygonZkEVM.sol#L1050-L1062:

```
- if (msg.sender == tx.origin) {
-     // Getting the calldata from an EOA is easy so no need to put the `transactions` in the
↪  event
-     emit ForceBatch(lastForceBatch, lastGlobalExitRoot, msg.sender, "");
- } else {
-     // Getting internal transaction calldata is complicated (because it requires an archive
↪  node)
-     // Therefore it's worth it to put the `transactions` in the event, which is easy to query
-     emit ForceBatch(
-         lastForceBatch,
-         lastGlobalExitRoot,
-         msg.sender,
-         transactions
-     );
- }
+ emit ForceBatch(
+     lastForceBatch,
+     lastGlobalExitRoot,
+     msg.sender,
+     transactions
+ );
```

**Polygon:** Acknowledged. Current chains using the `PolygonRollupBaseEtrog.sol` don't have the EIP-7702 implemented and there is no plan to keep updating those chains but upgrading them to PP consensus where the force batch function is not implemented.

**Spearbit:** Acknowledged.

### 5.3.14 A single failure to `write_local_network_state` will DOS certificate processing for network in agglayer

**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** During `handle_certificate_settlement`:

1. The network task updates the state in memory:

   agglayer-certificate-orchestrator/src/network_task.rs#L674:

   ```
   self.local_state = new;
   ```

2. The network task writes the new roots and inserts new leaves into persisted storage:

   agglayer-certificate-orchestrator/src/network_task.rs#L683-L692:

   ```
   self.state_store
   .write_local_network_state(
       &certificate.network_id,
       &self.local_state,
       new_leaves.as_slice(),
   )
   .map_err(|e| Error::PersistenceError {
       certificate_id,
       error: e.to_string(),
   })?;
   ```

   We notice that if the call to `state_store.write_local_network_state` fails, the certificate is not put in `InError` state, but the `next_expected_height` is not updated either. As a consequence, legitimate certificates coming from Aggkit will be denied. Subsequently, if Agglayer is restarted, it will reconstruct the state before settlement.

   agglayer-storage/src/stores/state/mod.rs#L233-L242:

   ```
   let new_leaf_count = new_state.exit_tree.leaf_count();
   let start_leaf_count = new_leaf_count - new_leaves.len() as u32;

   if let Some(stored_exit_tree) = self.read_local_exit_tree(network_id.into())? {
       //@audit check will fail upon next attempt to
       if stored_exit_tree.leaf_count() != start_leaf_count {
           return Err(Error::InconsistentState {
               network_id: network_id.into(),
           });
       }
   }
   ```

**Impact:** Relatively high, since Agglayer would stop processing the certificates of L2 attached to it.

**Likelihood:** Very low, since it requires a very specific failure in a minimal section (writing to `rocksdb`).

**Recommendation:** Consider adding a procedure to rebuild state storage from the history of settled certificates.

**Polygon:** Acknowledged. This is a known limitation of our current storage design and implementation. We have a ticket to track the improvements to it, but considering the low likelihood and the fact it can't be attacker-controlled, we're probably not going to fix that short-term.

**Spearbit:** Acknowledged.

#### 5.3.15 `handle_candidate` should set `latest_settled` and `at_capacity_for_epoch` after `handle_certificate_settlement`

**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** We notice that during `handle_pending` flow, network task updates `latest_settled` and `at_capacity_for_epoch`, after a successful settlement:

agglayer-certificate-orchestrator/src/network_task.rs#L440-L450:

```
Ok(settled) => {
    self.latest_settled = Some(settled); // <<<
    *next_expected_height += 1;
    self.at_capacity_for_epoch = true; // <<<

    debug!(
        hash = certificate_id.to_string(),
        "Certification process completed for {certificate_id} for network {}",
        self.network_id
    );
}
```

Unfortunately, after the `handle_candidate` flow finishes, these variables are left unset.

**Impact:** The impact is low, since the fields `latest_settled` and `at_capacity_for_epoch` are only used to perform additional sanity checks.

**Recommendation:** We recommend updating `latest_settled` as currently done during `handle_pending` for the sake of state handling consistency:

agglayer-certificate-orchestrator/src/network_task.rs#L352-L359:

```
  info!(
      hash,
      "Recovered the transaction for certificate {}", certificate_id
  );

  self.handle_certificate_settlement(certificate)?;
  *next_expected_height += 1;
+ self.at_capacity_for_epoch = true
+ self.latest_settled = Some(settled_certificate);
```

**Polygon:** Fixed in PR 819.

**Spearbit:** Fix verified.

#### 5.3.16 Reusing signature to re-submit an `InError` certificate with a different id, will DoS `Aggsender`

**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** Currently, submitting a certificate to Agglayer is permissioned. Aggsender signs a commitment to a subset of the certificate data before sending to `Agglayer`. We notice that the protection against replaying such signatures is weak; The `height` and `exitRoot` are part of the signature and would prevent the certificate signature from being reused to be settled twice. However, in the case the certificate fails to settle and falls into `InError` state, the same certificate with the same signature can be resubmitted to the `Agglayer`. Even worse, the aggsender does not commit to some data in the certificate. Another user who got access to a signature previously used to send a certificate, but which ended up in `InError`, can alter these fields, registering a certificate with the same signature but with a different certificate id. As a result, the Aggkit would not be able to reconcile its local state with the Agglayer state:

aggkit/aggsender/aggsender.go#L394:

```
//@audit Aggsender would not be aware of the certificate that was sent externally by reusing the
↪  signature for a failed certificate
//@audit As a result, it would still attempt to send certificates for height H, whereas agglayer expects
↪  H+1
for _, certificateLocal := range pendingCertificates {
    certificateHeader, err := a.aggLayerClient.GetCertificateHeader(ctx, certificateLocal.CertificateID)
```

> An example of field which could be changed without requiring a new signature would be the l1_leaf provided in claim data for an imported bridge exit.

**Impact analysis:** The Aggsender would attempt to submit certificates for height H repeatedly, whereas the Agglayer would expect H+1, halting the processing of certificates for the network until manual intervention.

**Likelihood analysis:** The external malicious actor would need to get access to an old signature that was for a certificate which failed to settle.

**Recommendation:** It would be more reliable for Aggsender to commit to the certificate id, as it would ensure that for any certificate signed by Aggsender, the Aggsender would be aware of it.

**Polygon:** Fixed in PR 920.

**Spearbit:** Fix verified.

## 5.4 Informational

### 5.4.1 Known vulnerability in `golang-jwt`

**Severity:** Informational

**Context:** go.mod#L108

**Description:** The `golang-jwt` library is used in its `v4.5.1` version. However, a vulnerability for this version is known.

```
$ govulncheck

=== Symbol Results ===

Vulnerability #1: GO-2025-3553
    Excessive memory allocation during header parsing in
    github.com/golang-jwt/jwt
  More info: https://pkg.go.dev/vuln/GO-2025-3553
  Module: github.com/golang-jwt/jwt/v4
    Found in: github.com/golang-jwt/jwt/v4@v4.5.1
    Fixed in: github.com/golang-jwt/jwt/v4@v4.5.2
    Example traces found:
      #1: pprof/pprof.go:66:33: pprof.StartProfilingHTTPServer calls http.Server.Serve, which eventually
      ↪  calls jwt.Parser.ParseUnverified

Your code is affected by 1 vulnerability from 1 module.
```

**Recommendation:** To fix the issue, consider upgrading `golang-jwt` to the `v4.5.2` version. Moreover, it is recommended to add `govulncheck` as part of your Github workflows to be warned when known vulnerabilities (CVE) are found. Such workflow can be found on the `golang/govulncheck-action` repository.

**Polygon:** Fixed in commit 73df673.

**Spearbit:** Fixed.

### 5.4.2 Hardcoded GER address in aggchain proof may not be accurate

**Severity:** Informational

**Context:** mod.rs#L25, proof.rs#L64

**Description:** The aggchain proof computation currently hardcodes the L2 GER address to a constant value (i.e. `L2_GER_ADDR`). This constant may not be accurate for existing chains.

**Recommendation:** The GER address could be a parameter linked to the specific chain. Another way to fix this is to adapt the proof for existing chains.

**Polygon:** Acknowledged. We completely define the genesis of native chains that use our bridge as the native bridge. For those chains, the GER has always the same address. For our next updates, we plan to support "outpost" chains, which are chains that doesn't use our native bridge or contracts and are chains already running. We still have not defined how to gather GER address for this kind of chains. For the version in the scope, there is no conflict in hard-coding the address of the GER.

**Spearbit:** Acknowledged.

#### 5.4.3 SP1 contract call crates are not meant for production usage

**Severity:** Informational

**Context:** Cargo.toml#L120-L122

**Description:** The SP1 contract call crates should not be used for production purposes in the current state of the codebase. This includes the `sp1-cc-client-executor` and the `sp1-cc-host-executor`. These crates are available in the sp1-contract-call Github repository which indicates:

> This repository is not meant for production usage.

**Recommendation:** The SP1 contract call crates should not be used for production usage.

**Polygon:** Acknowledged. We plan to move an alternative solution using storage proofs: see protocol-research issue 159

**Spearbit:** Acknowledged.

#### 5.4.4 Gas token existence check is not complete

**Severity:** Informational

**Context:** PolygonZkEVMBridgeV2.sol#L451-L454

**Description:** The `bridgeMessage` function checks if a gas token exist by checking that the `WETHToken` address is not zero. However, it does not check that the `gasTokenAddress` is not zero.

```
// If exist a gas token, only allow call this function without value
if (msg.value != 0 && address(WETHToken) != address(0)) {
    revert NoValueInMessagesOnGasTokenNetworks();
}
```

**Recommendation:** `gasTokenAddress` variable should be checked.

```
if (msg.value != 0 && gasTokenAddress != address(0))
```

**Polygon:** Acknowledged. agglayer-contracts PR 501 is available but not yet merged.

**Spearbit:** Acknowledged.

#### 5.4.5 Asset bridging does not allow identifying the source address

**Severity:** Informational

**Context:** PolygonZkEVMBridgeV2.sol#L410-L419

**Description:** The current implementation of asset bridging does not allow to encode any additional information from the source chain such as the source sender of the tokens. This can be inconvenient for projects building on top of Agglayer.

**Recommendation:** Consider adding the token sender (i.e. `msg.sender`) as part of the `metadata`.

**Polygon:** Acknowledged. The leave on the LocalExitTree does not need this informatin since the receiver could be anyone on the destination chain. This is commonly used in bridges. If the user do not set provide "advanced

data / custom data", the bridge UI will sent to the same address as the msg.sender. Howvere, the user has the ability to set a different receiver on the destination chain. We think there is no security impact. `msg.sender` can be retrieved from the event if needed, since it is straightforward to get the `tx.hash`.

**Spearbit:** Acknowledged. This finding does not have security impact.

### 5.4.6 AggKit listens to incorrect `VerifyBatches` events

**Severity:** Informational

**Context:** downloader.go#L20-L25

**Description:** The `VerifyBatches` event is defined as `VerifyBatches(uint32,uint64,bytes32,bytes32,address)` in the AggKit codebase. However, in the `v2/lib/PolygonRollupBaseEtrog.sol` file, this event is defined as `VerifyBatches(uint64,bytes32,address)`. The `VerifyBatches` event is only used by the deprecated `PolygonZkEVM` contract and by the `PolygonRollupBaseEtrog` contract. Any time a `VerifyBatches` event is emitted, a `VerifyBatchesTrustedAggregator` event is also emitted. Currently, AggKit will only parse the second one. As such, the AggKit is not expected to parse the `VerifyBatches` event.

**Recommendation:** The logic for `VerifyBatchesTrustedAggregator` and `VerifyBatches` events is the same. Consider removing the `VerifyBatches` event from the ones listened by the AggKit.

**Polygon:** Fixed in aggkit PR 693.

**Spearbit:** Fixed. The deprecated `VerifyBatches` event is not parsed by the AggKit anymore.

### 5.4.7 Removed log processing in the EVM downloader can be optimized

**Severity:** Informational

**Context:** evmdownloader.go#L390-L400

**Description:** The removed logs are still processed in `EVMDownloaderImplementation.GetLogs`.

```
for _, l := range unfilteredLogs {
    if l.Removed { // @POC: removed field checked once
        d.log.Warnf("log removed: %+v", l)
    }
    for _, topic := range d.topicsToQuery {
        if l.Topics[0] == topic && !l.Removed { // @POC: removed field checked twice
            logs = append(logs, l)
            break
        }
    }
}
```

**Recommendation:** The removed logs can be ignored by using `continue`.

```
  for _, l := range unfilteredLogs {
      if l.Removed {
          d.log.Warnf("log removed: %+v", l)
+         continue
      }
      for _, topic := range d.topicsToQuery {
          if l.Topics[0] == topic && !l.Removed {
              logs = append(logs, l)
              break
          }
      }
  }
```

**Polygon:** Fixed in aggkit PR 222.

**Spearbit:** Fixed. A fix similar to the recommended one has been applied.

#### 5.4.8 Shadowed error prevents rollback

**Severity:** Informational

**Context:** processor.go#L263

**Description:** The `err` variable is shadowed here. If the `tx.Commit()` fails, the `defer func()` will not see the error and will not call `tx.Rollback()`.

**Recommendation:** Do not shadow the `err` variable with the `tx.Commit()` return value, or implement `shouldRollback` logic similar to `ProcessBlock`.

See aggkit/bridgesync/processor.go#L281-L288:

```
shouldRollback := true
defer func() {
    if shouldRollback {
        if errRllbck := tx.Rollback(); errRllbck != nil {
            log.Errorf("error while rolling back tx %v", errRllbck)
        }
    }
}()
```

**Polygon:** Fixed in PR 544.

**Spearbit:** Fix verified.

#### 5.4.9 `subscriptions` race condition in `ReorgDetector` at startup

**Severity:** Informational

**Context:** reorgdetector.go#L270

**Description:** The `subscriptions` write at aggkit/reorgdetector/reorgdetector.go#L269-L274 isn't protected by the `subscriptionsLock`. A race condition is unlikely but possible, because the `ReorgDetector.Start()` function is started in a goroutine at aggkit/cmd/run.go#L478-L480 This is only possible once at startup. Still, a race at startup could lead to undefined behavior around the subscription.

**Recommendation:** Protect the `rd.subscriptions` write with the `rd.subscriptionsLock`.

**Polygon:** Fixed in aggkit PR 682.

**Spearbit:** Fix verified.

#### 5.4.10 GRPC client for AggKit uses insecure transport

**Severity:** Informational

**Context:** agglayer_grpc_client.go#L30-L45, grpc_client.go#L22

**Description:** There are two gRPC clients in AggKit that utilize the base gRPC client configuration - `AgglayerGRPCClient` and `AggchainProofClient`. Currently, the gRPC calls utilize no credentials over HTTP. This means that all calls are in plaintext and easily intercepted or modified.

**Recommendation:** Add the option for the gRPC client to use TLS credentials.

**Polygon:** Fixed in PR 222.

**Spearbit:** Fix verified.

### 5.4.11 `GetLatestInfoUntilBlock` does not guarantee returning Info within defined block limit

**Severity:** Informational

**Context:** processor.go#L179-L215

**Description:** The `GetLatestInfoUntilBlock` function at aggkit/l1infotreesync/processor.go#L179-L215 grabs an L1InfoTreeLeaf from the sql DB. In the meddler query, it orders by the block number and block position, always grabbing the latest leaf:

```
SELECT * FROM l1info_leaf ORDER BY block_num DESC, block_pos DESC LIMIT 1;
```

If the `blockNum` is equal to the last processed block (`lpb`), this is fine. If the `blockNum` is less than `lpb`, then this function will still return the latest info from the `lpb` instead of returning the latest info before `blockNum`. In the current usage of this function, this behavior does not create any problems.

**Recommendation:** The sql query should include a `WHERE` clause that limits the returned leafs:

```
SELECT * FROM l1info_leaf WHERE block_num <= $1 ORDER BY block_num DESC, block_pos DESC LIMIT 1;
```

**Polygon:** Fixed in PR 720.

**Spearbit:** Fix verified.

### 5.4.12 Structure size estimation is obsolete

**Severity:** Informational

**Context:** certificate_build_params.go#L101-L118

**Description:** The `CertificateBuildParams.EstimatedSize` estimates the size of the structure for the JSON format. However, the protobuf format is now used with gRPC. This makes the estimation inaccurate.

**Recommendation:** Consider reviewing the estimation for gRPC communications.

**Polygon:** Fixed in aggkit PR 564.

**Spearbit:** Fixed. The estimation has been reviewed to match gRPC.

### 5.4.13 Hardcoded range vkey in aggkit-prover reduces flexibility for upgrading

**Severity:** Informational

**Context:** lib.rs#L288

**Description:** The range proving program, which is responsible for proving the state transition between an `agreed_l2_output` and `claimed_l2_output`, has an associated vkey which is stored separately in three different places:

1. Aggkit prover (aggchain-proof-builder/src/lib.rs#L288):

   ```
   range_vkey_commitment: RANGE_VKEY_COMMITMENT,
   ```

2. Succinct-proposer (succinctlabs/op-succinct/blob/main/validity/src/proposer.rs#L98):

   ```
   let (range_pk, range_vk) = network_prover.setup(get_range_elf_embedded());
   ```

3. AggchainFEP (agglayer-contracts/contracts/v2/aggchains/AggchainFEP.sol#L86-L88):

   ```
   /// @notice The 32 byte commitment to the BabyBear representation of the verification key of the
   ↪  range SP1 program. Specifically,
   /// this verification is the output of converting the [u32; 8] range BabyBear verification key to
   ↪  a [u8; 32] array.
   bytes32 public rangeVkeyCommitment;
   ```

**Impact:** If the rangeVKey is updated in the contracts when using `updateRangeVkeyCommitment`, the current setup will fail to catch the mismatch between the prover network and on-chain parameter unless all of the values are updated.

**Recommendation:** Please consider using an RPC-retrieved value instead of a hardcoded one.

> One could fallback to the dummy default value in case the `FepVerification` is of type `ECDSA`.

**Polygon:** Acknowledged, but low priority. No target version yet.

**Spearbit:** Acknowledged.

#### 5.4.14 A failing network task will not restart in case of storage failure

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** When a network task encounters a `DBError` when handling an ended epoch, it may stop early (the future finishes with a success):

- Network task stopping processing incoming certificates:

  agglayer-certificate-orchestrator/src/lib.rs#L401-L407:

```
Poll::Ready(Some(Err(error))) => {
    warn!(
        "Network task Critical error during p-proof generation: {:?}",
        error
    );
    // TODO: Need to find a way to remove the task
}
```

- DBError is converted to InternalError:

  agglayer-certificate-orchestrator/src/lib.rs#L326-L334:

```
agglayer_storage::error::Error::DBError(error) => {
    let msg = format!(
        "CRITICAL error during packing of epoch {}: {}",
        epoch, error
    );
    error!(msg);
    self.cancellation_token.cancel();
    return Err(Error::InternalError(msg));
}
```

In this case, however, if a new certificate is sent for the network, a new attempt to respawn the network worker will be a noop. This is because the network task is never cleaned from the `spawned_network_tasks` set (agglayer-certificate-orchestrator/src/lib.rs#L253-L259):

```
fn spawn_network_task(&mut self, network_id: NetworkId) -> Result<(), Error> {
    //@audit network task was never cleaned from this set
    if self.spawned_network_tasks.contains_key(&network_id) {
        debug!("Network task already spawned for network {}", network_id);

        return Ok(());
    }
```

**Recommendation:** The Polygon team is aware of the issue, and as per the already present comment, this issue should be solved by removing the network task from the `spawned_network_tasks` in case of failure.

**Polygon:** Fixed in agglayer PR 812.

**Spearbit:** Fix verified.

#### 5.4.15 `wrappedAddressIsNotMintable` can be used to censor claims

**Severity:** Informational

**Context:** BridgeL2SovereignChain.sol#L832-L852

**Description:** The `_claimWrappedAsset` function will execute a transfer token instead of a token mint when the `wrappedAddressIsNotMintable` variable for the token is set to `true`. This allows the bridge manager to censor claims based on the token. When the token is marked as non-mintable and the balance is zero, the claims using this token will not be claimable.

**Recommendation:** This functionality is only accessible to the bridge manager. This censorship vector is not avoidable when the "not mintable" functionality is available.

**Polygon:** Acknowledged. `bridgeManager` is the only role that can alter the `wrappedAddressIsNotMintable` map. This role is controlled by us. Also, the mapping/mintable features will be removed in the upcoming versions, as they have been replaced by the "upgradeableWrappedToken".

**Spearbit:** Acknowledged.

#### 5.4.16 Recursive function leads to high memory consumption on error

**Severity:** Informational

**Context:** evmdownloader.go#L305-L326

**Description:** The `GetEventsByBlockRange` function calls itself when a block hash mismatch is found. However, in case of a persistent mismatch, this function will reenter itself indefinitely until the memory consumption is so high that the process is killed by the operating system.

```
func (d *EVMDownloaderImplementation) GetEventsByBlockRange(ctx context.Context, fromBlock, toBlock
↪  uint64) EVMBlocks {
    select {
    case <-ctx.Done():
        return nil
    default:
        blocks := EVMBlocks{}
        logs := d.GetLogs(ctx, fromBlock, toBlock)
        for _, l := range logs {
            if len(blocks) == 0 || blocks[len(blocks)-1].Num < l.BlockNumber {
                // ...

                if b.Hash != l.BlockHash {
                    // ...
                    return d.GetEventsByBlockRange(ctx, fromBlock, toBlock) // @audit recursive call
                }
```

**Recommendation:** Consider exiting the process properly when a persistent mismatch is found. For example, this can be done after re-entering the function X times.

**Polygon:** Fixed in aggkit PR 672.

**Spearbit:** Fixed. A counter is implemented and will stop execution at a given maximum.

#### 5.4.17 Missing zero address check for non-existing rollups

**Severity:** Informational

**Context:** lib.rs#L302-L322

**Description:** The Agglayer can receive certificates for non-existing chains. The `netword_id` of the certificate is used to retrieve the trusted sequencer from the rollup data on L1. This is achieved in `verify_cert_signature()` by calling `get_trusted_sequencer_address()`. However, this function doesn't explicitly revert on a zero config which is the default when a chain doesn't exist in the rollup manager. A call will be made to `PolygonZkEVM(0x00).trustedSequencer()`. This will revert as no address is returned.

```
async fn get_trusted_sequencer_address(
    &self,
    rollup_id: u32,
    proof_signers: HashMap<u32, Address>,
) -> Result<Address, L1RpcError> {
    if let Some(addr) = proof_signers.get(&rollup_id) {
        Ok(*addr)
    } else {
        let rollup_data = self
            .inner
            .rollup_id_to_rollup_data(rollup_id)
            .await
            .map_err(|_| L1RpcError::RollupDataRetrievalFailed)?;

        let rollup_metadata = RollupIDToRollupDataReturn { rollup_data };
        PolygonZkEvm::new(
            rollup_metadata.rollup_data.rollup_contract, // @audit this is not checked for zero address
            self.inner.client().clone(),
        )
        .trusted_sequencer()
        .await
        .map_err(|_| L1RpcError::TrustedSequencerRetrievalFailed)
    }
}
```

**Recommendation:** Consider checking that `rollup_metadata.rollup_data.rollup_contract` is not zero to avoid making a call to address zero.

**Polygon:** Fixed in agglayer PR 928.

**Spearbit:** Fixed. Zero address check has been added.

#### 5.4.18 Agglayer discards local native execution public values for pessimistic proof

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** In Agglayer, during the `certify` call, the network task computes the request to send to the pessimistic prover by locally executing `witness_execution`. However, as we can see, the public values computed during native `generate_pessimistic_proof` execution are ignored (certifier/mod.rs#L414-L415):

```
let _ = generate_pessimistic_proof(initial_state.clone().into(), &multi_batch_header)
    .map_err(|source| CertificationError::NativeExecutionFailed { source })?;
```

**Impact:** The prover should not be able to manipulate some public values (due to proof verification and state validation in contracts), and it would be unlikely to behave maliciously, since it should be hosted in the same environment as agglayer; It would be a simple additional caution to check that values from the `PessimisticProofOutput` match the ones used by the certificate.

**Recommendation:** Implement a check similar to aggkit-prover (aggchain-proof-builder/src/lib.rs#L297-L310):

```
let retrieved_from_contracts = AggregationProofPublicValues::from(&fep_inputs);

if aggregation_proof_public_values != &retrieved_from_contracts {
    error!(
        "Mismatch between the aggregation proof public values - retrieved \
            from the contracts: {retrieved_from_contracts:?}, received with the \
            proof: {:?}",
        aggregation_proof_public_values
    );
    return Err(Error::MismatchAggregationProofPublicValues {
        expected_by_contract: Box::new(retrieved_from_contracts),
        expected_by_verifier: Box::new(aggregation_proof_public_values.clone()),
    });
}
```

**Polygon:** Fixed in PR 771.

**Spearbit:** Fix verified.

#### 5.4.19 Incorrect field name for aggchainData in certificate json serialization

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** In the annotations used to serialize the certificate to json in aggkit, the `aggchainData` field is serialized to `data`, whereas `aggchain_data` should be used (agglayer/types/types.go#L239-L264):

```
type Certificate struct {
    NetworkID           uint32                `json:"network_id"`
    Height              uint64                `json:"height"`
    PrevLocalExitRoot   common.Hash           `json:"prev_local_exit_root"`
    NewLocalExitRoot    common.Hash           `json:"new_local_exit_root"`
    BridgeExits         []*BridgeExit         `json:"bridge_exits"`
    ImportedBridgeExits []*ImportedBridgeExit `json:"imported_bridge_exits"`
    Metadata            common.Hash           `json:"metadata"`
    CustomChainData     []byte                `json:"custom_chain_data,omitempty"`
    //@audit should be aggchain_data
    AggchainData        AggchainData          `json:"data,omitempty"`
    L1InfoTreeLeafCount uint32                `json:"l1_info_tree_leaf_count,omitempty"`
}

// UnmarshalJSON is the implementation of the json.Unmarshaler interface
func (c *Certificate) UnmarshalJSON(data []byte) error {
    aux := &struct {
        NetworkID           uint32                `json:"network_id"`
        Height              uint64                `json:"height"`
        PrevLocalExitRoot   common.Hash           `json:"prev_local_exit_root"`
        NewLocalExitRoot    common.Hash           `json:"new_local_exit_root"`
        BridgeExits         []*BridgeExit         `json:"bridge_exits"`
        ImportedBridgeExits []*ImportedBridgeExit `json:"imported_bridge_exits"`
        Metadata            common.Hash           `json:"metadata"`
        CustomChainData     []byte                `json:"custom_chain_data,omitempty"`
        //@audit should be aggchain_data
        AggchainData        AggchainDataSelector  `json:"data,omitempty"`
        L1InfoTreeLeafCount uint32                `json:"l1_info_tree_leaf_count,omitempty"`
    }{}
```

However, the impact is very limited, because gRPC is now used for communication between aggkit and agglayer;

**Recommendation:** Consider aligning the naming on agglayer:

```
  type Certificate struct {
    NetworkID           uint32               `json:"network_id"`
    Height              uint64               `json:"height"`
    PrevLocalExitRoot   common.Hash          `json:"prev_local_exit_root"`
    NewLocalExitRoot    common.Hash          `json:"new_local_exit_root"`
    BridgeExits         []*BridgeExit        `json:"bridge_exits"`
    ImportedBridgeExits []*ImportedBridgeExit `json:"imported_bridge_exits"`
    Metadata            common.Hash          `json:"metadata"`
    CustomChainData     []byte               `json:"custom_chain_data,omitempty"`
-   AggchainData        AggchainData         `json:"data,omitempty"`
+   AggchainData        AggchainData         `json:"aggchain_data,omitempty"`
    L1InfoTreeLeafCount uint32               `json:"l1_info_tree_leaf_count,omitempty"`
  }

  // UnmarshalJSON is the implementation of the json.Unmarshaler interface
  func (c *Certificate) UnmarshalJSON(data []byte) error {
    aux := &struct {
        NetworkID           uint32               `json:"network_id"`
        Height              uint64               `json:"height"`
        PrevLocalExitRoot   common.Hash          `json:"prev_local_exit_root"`
        NewLocalExitRoot    common.Hash          `json:"new_local_exit_root"`
        BridgeExits         []*BridgeExit        `json:"bridge_exits"`
        ImportedBridgeExits []*ImportedBridgeExit `json:"imported_bridge_exits"`
        Metadata            common.Hash          `json:"metadata"`
        CustomChainData     []byte               `json:"custom_chain_data,omitempty"`
-       AggchainData        AggchainDataSelector `json:"data,omitempty"`
+       AggchainData        AggchainDataSelector `json:"aggchain_data,omitempty"`
        L1InfoTreeLeafCount uint32               `json:"l1_info_tree_leaf_count,omitempty"`
    }{}
```

**Polygon:** Fixed in types.go#L258.

**Spearbit:** Fix verified.

#### 5.4.20 Op-Succinct aggregation proof generation may fail due to old L1 block hash

**Severity:** Informational

**Context:** service.rs#L151-L169

**Description:** The AggKit will generate an Aggchain proof that includes the Op-Succinct aggregation proof for L2 blocks that are finalized. In the context of Op-Succint, "finalized" means that the L2 blocks are available on the DA layer. However when the last finalized L1 info tree leaf is old (i.e. it has not been updated since a long time, because there was no activity for example), then the Op-Succinct aggregation proof may not be provable for the given L2 block range against the given L1 block hash. This is because the transaction that includes L2 blocks data (e.g. blobs) has been included in a block after the L1 block hash corresponding to the last finalized L1 info tree leaf. The Op-Succinct aggregation proof generation will fail because the data is not available on Ethereum at this block.

**Recommendation:** This denial of service vector arises when the system does not have enough activity. As long as there is enough activity, the block hash in the L1 info tree leaf will be recent enough for the data to be available. An additional mechanism to update the L1 info tree leaf could be implemented.

**Polygon:** Acknowledged. We plan on developing a bot that forces an update if there is no "natural activity" for more than X minutes.

**Spearbit:** Acknowledged. This issue is by design, having a bot to forge updates if needed is a reliable solution.

#### 5.4.21 Empty metadata are handled differently than non-empty

**Severity:** Informational

**Context:** types.go#L467-L478, flow_base.go#L218-L234

**Description:** The `convertBridgeMetadata` function handles metadata differently when these are empty.

```
func convertBridgeMetadata(metadata []byte, importedBridgeMetadataAsHash bool) ([]byte, bool) {
    var (
        metaData         []byte
        isMetadataHashed bool
    )

    if importedBridgeMetadataAsHash && len(metadata) > 0 {
        metaData = crypto.Keccak256(metadata)
        isMetadataHashed = true
    } else {
        metaData = metadata
        isMetadataHashed = false
    }
```

Fortunately, the `BridgeExit.Hash` function handles the edge case but this is highly confusing.

```
func (b *BridgeExit) Hash() common.Hash {
    if b.Amount == nil {
        b.Amount = big.NewInt(0)
    }
    var metaDataHash []byte
    if b.IsMetadataHashed {
        metaDataHash = b.Metadata
    } else {
        metaDataHash = crypto.Keccak256(b.Metadata)
    }
```

**Recommendation:** Consider handling empty and non-empty metadata in the same way.

**Polygon:** Fixed in agglayer issue 707.

**Spearbit:** Fixed. The two cases are handled in the same way.

#### 5.4.22 Typos, renaming suggestions, unused variables

**Severity:** Informational

**Context:** oracle.go#L17

**Description:**

1. Rename L1InfoTreer to `L1InfoTreeSyncer`.
2. Use consistent casing in AggchainParamValues.
3. pending_state field is not needed, its uses can be replaced by using a local (non-optional variable). Indeed, all uses are preceded by an initialization to Some:
   - network_task.rs#L615.
   - network_task.rs#L343.

**Polygon:** Fixed in aggkit PR 742, provers PR 256 and agglayer PR 932.

**Spearbit:** Fix verified.

#### 5.4.23 `debug_traceTransaction` not intended for production use

**Severity:** Informational

**Context:** downloader.go#L149

**Description:** The `debug_traceTransaction` API call requires historical state and must be able to re-execute the transaction. The connected "EthClient" must retain historical state for as long as the bridgesync requires for the claim, even with slowdown from unexpected circumstances.

Additionally, the `debug_*` API endpoints are not designed for production use. They are not specified in Ethereum execution APIs, are client-specific, and are prone to change. While we do not expect Geth's `debug_traceTransaction` API call to have any breaking changes, there is no standardization that guarantees stability and backwards compatibility of this API endpoint.

**Recommendation:** The `debug_traceTransaction` call is used to gather parts of the calldata from the `claim*` call that corresponds to a specific `ClaimEvent`. This API call should be avoided in AggKit if possible. This can be done with some restructuring by placing the calldata into the `ClaimEvent` or storing the data on chain.

If restructuring is not an option, then we must ensure that the connected Geth client stores the historical state required for a claim. Because of the possibility of slowdown, crashes, or unexpected circumstances, we recommend that the Geth client be configured as an archival node. This will ensure that the Geth client retains the required state to make the `debug_traceTransaction` call.

**Polygon:** Acknowledged.

**Spearbit:** Acknowledged.