Polygon

AggLayer v0.3.0 - Offchain Updates

Security Assessment Report

Version: 1.0

April, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Polygon components. The review focused solely on the security aspects of the Rust implementation of the components in scope, though general recommendations and informational comments are also provided.

Disclaimer

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

Document Structure

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

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

Overview

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

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

The AggLayer v0.3.0 update specifically introduces support for pessimistic proofs. To achieve this, updates have been made to the Rust crates that make up the pessimistic proof system for Agglayer. In addition to this, updates have been made to the prover system and some other Agglayer crates which were also covered in this audit of AggLayer v0.3.0.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the agglayer/agglayer repository.

The scope of this time-boxed review was strictly limited to files at tag v0.3.0-rc.6 and specifically the following directories:

- `crates/pessimistic-proof-program/`
- `crates/pessimistic-proof-core/`
- `crates/pessimistic-proof/`

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

Approach

The security assessment covered components written in Rust.

The manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team also utilised the following automated testing tools:

- Clippy linting: `https://doc.rust-lang.org/stable/clippy/index.html`
- Cargo Audit: `https://github.com/RustSec/rustsec/tree/main/cargo-audit`
- Cargo Outdated: `https://github.com/kbknapp/cargo-outdated`
- Cargo Geiger: `https://github.com/rust-secure-code/cargo-geiger`
- Cargo Tarpaulin: `https://crates.io/crates/cargo-tarpaulin`

Output for these automated tools is available upon request.

Coverage Limitations

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

Findings Summary

The testing team identified a total of 8 issues during this assessment. Categorised by their severity:

- Medium: 2 issues.
- Low: 3 issues.
- Informational: 3 issues.

Detailed Findings

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

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

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

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

Summary of Findings

ID	Description	Severity	Status
AGLO3-01	Use Of Unhashed Leaf Values In Merkle Tree	Medium	Closed
AGLO3-02	Use of `H::Digest::default()` for Empty Nodes May Break Non-Inclusion Logic	Medium	Closed
AGLO3-03	Field Omission in L1 Leaf Hash	Low	Open
AGLO3-04	Mistaken Network Identity In The Event Of Integer Overflow	Low	Open
AGLO3-05	Unchecked Use of `unwrap()` May Cause Panics	Low	Open
AGLO3-06	Unchecked `TREE_DEPTH` Values May Cause Compile-Time Panics And Logic Issues	Informational	Open
AGLO3-07	Potential Cross-Structs Keccak Hash Collisions	Informational	Open
AGLO3-08	Miscellaneous General Comments	Informational	Open

AGLO3-01	Use Of Unhashed Leaf Values In Merkle Tree		
Asset	`crates/pessimistic-proof-core/src/utils/smt.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Using `U256` for `Digest` implies a conversion from a number to a cryptographic hash, but in the current implementation, `FromU256()` populates `Digest` with raw `U256` bytes instead of a cryptographic hash, breaking the assumption that `Digest` is a hash commitment:

```
impl FromU256 for Digest {
    fn from_u256(u: U256) -> Self {
        Self(u.to_be_bytes())
    }
}
```

This is used in `verify_and_update()` in `crates/pessimistic-proof-core/src/local_balance_tree.rs`, which converts `old_value` and `new_value` into raw byte form:

```
pub fn verify_and_update(
    &mut self,
    key: TokenInfo,
    path_to_update: &LocalBalancePath,
    old_balance: U256,
    new_balance: U256,
) -> Result<(), ProofError> {
    self.root = path_to_update
        .verify_and_update(
            key,
            H::Digest::from_u256(old_balance),  // @audit raw bytes, not hash
            H::Digest::from_u256(new_balance),  //
            self.root,
        )
        .ok_or(ProofError::InvalidBalancePath)?;

    Ok(())
}
```

Which is then used in the following implementation in `crates/pessimistic-proof-core/src/utils/smt.rs`:

```
pub fn verify_and_update(
        &self,
        key: K,
        old_value: H::Digest,
        new_value: H::Digest,
        root: H::Digest,
    ) -> Option
    where
        K: ToBits + Copy,
    {
        if !self.verify(key, old_value, root) {
            return None;
        }
        let bits = key.to_bits();
        let mut hash = new_value;   // @audit this is not a hash, but raw bytes value
        for i in 0..DEPTH {
            hash = if bits[DEPTH - i - 1] {
                H::merge(&self.siblings[i], &hash)
            } else {
                H::merge(&hash, &self.siblings[i])
            };
        }

        Some(hash)
    }
```

In standard Merkle tree designs, leaves are hashed to cryptographically bind data to the tree. If raw values (e.g. `U256` bytes) are used instead, an attacker could construct valid proofs with arbitrary inputs, as no preimage is required. This breaks the integrity guarantees of the tree and can lead to proof forgery in systems that assume hashed leaves.

Recommendations

Instead of accepting `old_value` and `new_value` as raw bytes, ensure that `H::Digest` actually represents a valid cryptographic hash, or hash the values of `old_value` and `new_value` inside the function before proceeding with verification.

E.g., redefine `FromU256` to hash the `U256` value:

```
impl FromU256 for Digest {
    fn from_u256(u: U256) -> Self {
        keccak256(u.to_be_bytes().as_slice())  // @audit, hash, not raw bytes
    }
}
```

Or, hash within `verify_and_update()`:

```
self.root = path_to_update.verify_and_update(
    key,
    H::hash(&old_balance.to_be_bytes()),  // @audit, hash explicitly
    H::hash(&new_balance.to_be_bytes()),
    self.root,
    ).ok_or(ProofError::InvalidBalancePath)?;
```

Resolution

After conversations with development team, this approach has been deemed valid by design within a resource constrained, SP1-based zero-knowledge environment where hashing is expensive, tree depth is fixed and privacy is not required.

The position of each node is structurally determined by the key, so there is no ambiguity between leaf and internal nodes. The system does not rely on the cryptographic binding of the leaf value itself but instead, on the overall integrity of the tree structure, which is preserved through secure hashing of internal nodes. This design avoids an extra hashing round at the leaf level, which is a practical optimisation given the cost of hashing in SP1-based environments.

As such, this finding has been closed based on the following comment from the development team:

> *"Here we're just having the leaves be encoded directly in the tree, rather than first hashed and then the hash being put in the tree. As our Merkle trees are fixed-size, I can't see us ever mistaking a leaf for an inner node, and this saves us one hashing round, which is pretty expensive on SP1."*

AGLO3-02	Use of `H::Digest::default()` for Empty Nodes May Break Non-Inclusion Logic		
Asset	`crates/pessimistic-proof/src/utils/smt.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The Merkle tree implementation uses `H::Digest::default()`, which resolves to `[0u8; 32]`, as the empty node value at each depth, forming the base of `empty_hash_at_height`. This value is used during non-inclusion proof generation to identify when a path in the tree is empty and terminate traversal early.

However, because leaf values are stored as raw `U256` bytes (e.g., `U256::to_be_bytes()`), a valid balance of zero will be encoded as `[0u8; 32]`, the same value as the designated empty digest. This makes a key with a zero balance indistinguishable from an unset key.

As a result, `get_non_inclusion_proof()` may return a valid-looking non-inclusion proof for a key that is in fact present in the tree with a zero balance. This violates the integrity of the tree and can lead to logic errors or exploitation if the application treats presence and absence differently.

Recommendations

Replace `H::Digest::default()` with a dedicated value that cannot be confused with valid leaf data, e.g. `H(b"EMPTY_LEAF")`.

Resolution

This finding has been closed with the following comment from the development team:

> *"Currently, all our trees have a known, fixed depth. The LBT is 192-bit deep, with 32 bits of network id and 160 bits of token address.*
>
> *So any node that is not 192-bit deep is an internal node, and any node that is exactly 192-bit deep is a leaf. And our trees are basically full: it's on purpose that all leafs default to 0, because the chain starts with a 0 balance for each token."*

AGLO3-03	Field Omission in L1 Leaf Hash		
Asset	`agglayer/crates/pessimistic-proof-core/src/imported_bridge_exit.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `L1InfoTreeLeaf::hash()` function derives its hash solely from the `L1InfoTreeLeafInner` fields (`global_exit_root` , `block_hash` , and `timestamp`). The outer fields `rer` and `mer` are excluded from the hash.

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct L1InfoTreeLeafInner {
    pub global_exit_root: Digest,
    pub block_hash: Digest,
    pub timestamp: u64,
}

impl L1InfoTreeLeafInner {
    pub fn hash(&self) -> Digest {
        keccak256_combine([
            self.global_exit_root.as_slice(),
            self.block_hash.as_slice(),
            &self.timestamp.to_be_bytes(),
        ])
    }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct L1InfoTreeLeaf {
    pub l1_info_tree_index: u32,
    pub rer: Digest,
    pub mer: Digest,
    pub inner: L1InfoTreeLeafInner,
}

impl L1InfoTreeLeaf {
    pub fn hash(&self) -> Digest {
        self.inner.hash()
    }
}
```

As a result, two distinct leaf structures with different `rer` or `mer` values may produce identical hashes. This breaks the cryptographic expectation that the hash uniquely represents the full content of the leaf.

Note, the `verify()` logic separately checks `rer` and `mer` via Merkle proofs, however, this reliance on out-of-band verification may weaken the Merkle tree's integrity guarantees.

Recommendations

Consider including all fields in the hash computation to ensure it reflects the full leaf state.

Alternatively, clarify in documentation or comments that the `hash()` function only commits to the `L1InfoTreeLeafInner`

subset, and that `rer` and `mer` are verified independently. This could help prevent misuse or incorrect assumptions about what is covered by the Merkle tree.

AGLO3-04	Mistaken Network Identity In The Event Of Integer Overflow		
Asset	`agglayer/crates/pessimistic-proof-core/src/global_index.rs`		
Status	Open		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When returning `network_id()`, it is possible for the value to overflow and return 0, which would misidentify a rollup network as the Ethereum mainnet:

```
pub fn network_id(&self) -> NetworkId {
    if self.mainnet_flag {
        0
    } else {
        self.rollup_index + 1
    }
}
```

This is possible because `rollup_index` is a `u32` value and so has a maximal value of `rollup_index = u32::MAX` (4,294,967,295). Returning `self.rollup_index + 1`, which is not a checked operation, will result in an integer overflow when `self.rollup_index == u32::MAX`.

Note, in practice, it is very unlikely the network IDs will reach this max index value, hence this issue has been rated as low risk.

Recommendations

Use checked addition or implement a range check to prevent a silent integer overflow from occurring.

AGLO3-05	Unchecked Use of `unwrap()` May Cause Panics		
Asset	`/pessimistic-proof-program/src/main.rs, /pessimistic-proof-core/src/global_index.rs`		
Status	Open		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Multiple instances of `unwrap()` were observed in the Pessimistic Proof program codebase, which may lead to panics if an error is encountered.

There were two instances identified in `main.rs`:

```
let outputs = generate_pessimistic_proof(initial_state, &batch_header).unwrap();

let pp_inputs = PessimisticProofOutput::bincode_options()
        .serialize(&outputs)
        .unwrap();
```

This could occur if `generate_pessimistic_proof()` fails, e.g. due to invalid roots or a block height overflow, or if `serialize()` fails, e.g. due to invalid data or issues with the serialisation process, such as memory exhaustion.

Instanced in `global_index.rs` on lines [**58-59**] were also observed:

```
let rollup_index = u32::from_le_bytes(bytes[4..8].try_into().unwrap());
let leaf_index = u32::from_le_bytes(bytes[0..4].try_into().unwrap());
```

This case is of lower risk as `bytes` is derived from `value.as_le_slice()`, which returns a 32-byte slice (as `U256` is 256 bits), making the slicing `0..4` and `4..8` safe. However, the use of `unwrap()` is still discouraged, as future changes to `as_le_slice()` could invalidate this assumption.

Recommendations

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

1. **Pattern matching**: Explicitly handle both `Some` / `None` and `Ok` / `Err` cases using match statements, ensuring all
possible cases are covered.
2. **`if let` or `while let` constructs**: These provide more concise handling of successful cases, with fallback behaviour in case of errors or missing values.
3. **Custom error handling**: Return relevant error messages or propagate errors using the `?` operator to gracefully handle failure scenarios, allowing errors to propagate up to higher levels of the application.

AGLO3-06	Unchecked `TREE_DEPTH` Values May Cause Compile-Time Panics And Logic Issues
Asset	`crates/pessimistic-proof-core/src/local_exit_tree/mod.rs, proof.rs`
Status	Open
Rating	Informational

Description

In `/pessimistic-proof-core/src/local_exit_tree/mod.rs` and `proof.rs`, the code assumes that `TREE_DEPTH <= 32`, but this is not enforced explicitly. If `TREE_DEPTH` is ever set above 32, it will result in compile-time panics and incorrect values, potentially leading to logic issues.

Specifically:

- In `mod.rs`, line [46]:

```
const MAX_NUM_LEAVES: u32 = ((1u64 << TREE_DEPTH) - 1) as u32;
```

If `TREE_DEPTH > 32`, the result of `1u64 << TREE_DEPTH` exceeds `u32::MAX`, and the cast to `u32` silently truncates the upper bits. This would result in an incorrect `MAX_NUM_LEAVES` value and could trigger a compile-time panic.

- In `proof.rs`, line [36]:

```
    pub siblings: [H::Digest; TREE_DEPTH],

(...)

impl LETMerkleProof
where
    H: Hasher,
    H::Digest: Eq + Copy + Default + Serialize + DeserializeOwned,
{
    pub fn verify(&self, leaf: H::Digest, leaf_index: u32, root: H::Digest) -> bool {
        let mut entry = leaf;
        let mut index = leaf_index;
        for &sibling in &self.siblings {
            entry = if index & 1 == 0 {
                H::merge(&entry, &sibling)
            } else {
                H::merge(&sibling, &entry)
            };
            index >>= 1;
        }
        if index != 0 {
            return false;
        }

        entry == root
    }
}
```

if `TREE_DEPTH > 32`, the index validation logic `index != 0` may behave incorrectly for large index values as `u32` can only store up to 32 bits, potentially allowing logically invalid proofs to be accepted under certain conditions.

While these issues are not currently exploitable as `TREE_DEPTH` is fixed to 32, they present a future maintenance risk.

Recommendations

Specifically for `crates/pessimistic-proof-core/src/local_exit_tree/proof.rs`, consider capping `TREE_DEPTH` to 32 or adding an explicit check, e.g.

```
if leaf_index >= 1 << TREE_DEPTH {
    return false;
}
```

AGLO3-07	Potential Cross-Structs Keccak Hash Collisions
Asset	`agglayer/crates/pessimistic-proof-core/src/keccak.rs`
Status	Open
Rating	Informational

Description

The `keccak256_combine()` function may produce identical hashes for distinct structs that have compatible total byte lengths but different internal layouts.

For example, the following two struct representations could yield the same hash if their raw bytes match:

- Struct A: `[u8; 4], [u8; 20]`
- Struct B: `[u8; 24]`

This could pose a risk in contexts where hash uniqueness is assumed for distinct data types or domains.

Recommendations

Consider introducing delimiters or a struct-specific domain separation by prefixing the input with a distinguishing label. For example:

```
hasher.update(b"BridgeExit");
```

AGLO3-08	Miscellaneous General Comments
Asset	All contracts
Status	Open
Rating	Informational

Description

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

1. **Unclear Bit Indexing In Specification Comment**

   Related Asset(s): crates/pessimistic-proof-core/src/global_index.rs

   The specification comment describing the `GlobalIndex` bit layout uses most-significant-bit (MSB) indexing, whereas the implementation uses least-significant-bit (LSB) indexing, as is conventional in Rust and byte-oriented programming.

   The comment defines the layout as:

   ```
   /// | 191 bits |    1 bit      |    32 bits    |    32 bits    |
   /// |    0     | mainnet flag  | rollup index  |  leaf index   |
   ```

   This implies that the mainnet flag resides at bit 191 (0-indexed from MSB). However, the implementation sets bit 64 (LSB-indexed):

   ```
   const MAINNET_FLAG_OFFSET: usize = 2 * 32; // bit 64 (LSB indexing)
   (...)
   bytes[8] |= 0x01; // sets bit 64
   ```

   This is consistent with the MSB-to-LSB mapping, as bit 191 (MSB) corresponds to bit 64 (LSB) in a 256-bit structure (255 - 191 = 64).

   Clarify the comment to explicitly state that the layout is described in MSB-first order, and that bit positions in code are interpreted in LSB-first order. Optionally, document both MSB and LSB views of the structure to improve clarity and prevent confusion.

2. **Address TODO Comments**

   Related Asset(s): agglayer/crates/pessimistic-proof-core/src/local_balance_tree.rs, agglayer/crates/pessimistic-proof-core/src/local_state/mod.rs

   Several files contain TODO comments outlining steps to be added to these files. In particular line [18] and line [40] of `local_balance_tree.rs` and line [92] and line [196] of `mod.rs`.

   Review these comments, removing and making code adjustments where necessary.

3. **Improve Error Context When Deserializing Hex-Encoded Digest**

   Related Asset(s): agglayer/crates/pessimistic-proof-core/src/keccak/digest.rs

   Hex parsing errors are forwarded directly via `serde::de::Error::custom(...)`, but without context (e.g., whether the length was wrong vs. invalid characters).

   Wrap the `from_hex` error to provide more context:

   ```
   let bytes = <[u8; 32]>::from_hex(s)
       .map_err(|e| serde::de::Error::custom(format!("invalid Digest hex: {}", e)))?;
   ```

4. **Deterministic Root Initialisation**

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

   Using a deterministic default root (e.g. `[0u8; 32]`) allows anyone to reproduce and verify the initial tree state, which is useful for auditability and stateless validation. While this predictability could theoretically enable pre-computation attacks, it is safe as long as all state transitions are verified and the root is never used to imply meaningful content before any insertions.

   Ensure that proofs are always checked against the current root and that an empty root is not treated as meaningful state.

5. **Debug Formatting Inconsistency**

   Related Asset(s): agglayer/crates/pessimistic-proof-core/src/keccak/digest.rs

   The `fmt::Debug` implementation should be more explicit, for example:

```
impl fmt::Debug for Digest {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Digest({:#x})", self)
    }
}
```

   Additionally, `Debug` and `Display` produce inconsistent outputs - `Display` returns a lowercase hexadecimal string prefixed with `0x`, while `Debug` does not. This inconsistency may cause confusion.

   Consider adding more details to the implementation of `fmt::Debug` for `Digest` and align string formatting in `Debug` and `Display` for consistency.

Recommendations

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

Appendix A Vulnerability Severity Classification

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

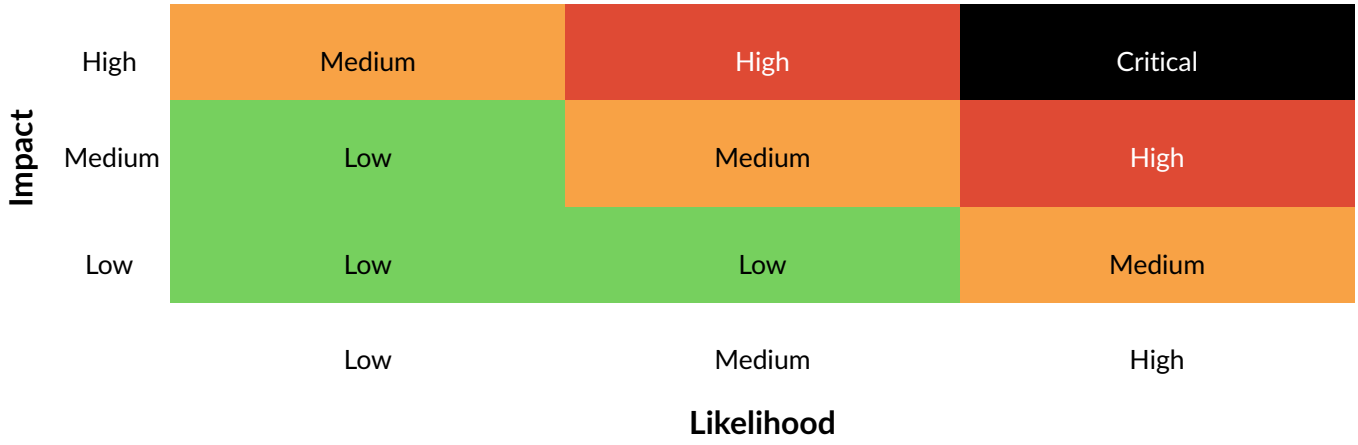

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References