Polygon

AggLayer v0.3.0 - Offchain Updates - Part 2

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 AggLayer 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 AggLayer 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 other AggLayer crates, which were also covered in this assessment.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the agglayer/agglayer and agglayer/provers repositories.

The scope of this time-boxed review was limited to the following crates from `agglayer` repository, assessed at tag v0.3.0-rc.7:

- `crates/agglayer-certificate-orchestrator`
- `crates/agglayer-storage`
- `crates/agglayer-grpc-api`
- `crates/agglayer-node`

For the `provers` repository, the scope of the review was limited to the following crates, assessed at tag v0.1.0-rc.4:

- `crates/aggchain-proof-program`
- `crates/prover-engine`
- `crates/prover-executor`

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 22 issues during this assessment. Categorised by their severity:

- Medium: 2 issues.
- Low: 12 issues.
- Informational: 8 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
AGLO3.2-01	Lack Of Authentication/Authorisation On RPC Services	Medium	Open
AGLO3.2-02	Network Tasks Can Drop Certificates If Channel Is Full	Medium	Open
AGLO3.2-03	Unencrypted Communication For RPC And Metric Endpoints	Low	Open
AGLO3.2-04	Blind Reproving Of Proven Certificates Without State Recovery Or Consistency Checks	Low	Open
AGLO3.2-05	Missing Recovery For Failed Network Tasks	Low	Open
AGLO3.2-06	Unchecked Panic With `unwrap()` And `expect()`	Low	Open
AGLO3.2-07	Use of `unwrap()` with gRPC reflection server can result in panic	Low	Open
AGLO3.2-08	`on_proven_certificate()` Continues On Partial Error	Low	Open
AGLO3.2-09	`insert_certificate_header()` Has No Conflict Detection For Settled Certificates	Low	Open
AGLO3.2-10	No Panic Handling On Orchestrator Or RPC Tasks	Low	Open
AGLO3.2-11	Potential Resource Leak On Panic In `LocalExecutor`	Low	Open
AGLO3.2-12	Potential Resource Leak On Panic Before Shutdown	Low	Open
AGLO3.2-13	`write_batch()` Skips `default_write_options`	Low	Open
AGLO3.2-14	Partial Error Handling In `create_new_backup()`	Low	Open
AGLO3.2-15	Unimplemented GPU Prover Type	Informational	Open
AGLO3.2-16	Fallback Mechanism Cloning Overhead	Informational	Open
AGLO3.2-17	Fallback Service Readiness Not Polled In Executor::poll_ready	Informational	Open
AGLO3.2-18	No Resource Cleanup At Shutdown	Informational	Open
AGLO3.2-19	Hardcoded Default Socket Addresses	Informational	Open
AGLO3.2-20	`LocalNetworkStateData` State Risks Incorrect State Recorded On Error Conditions	Informational	Open
AGLO3.2-21	Task Spawning Without Limits	Informational	Open
AGLO3.2-22	Miscellaneous General Comments	Informational	Open

AGLO3.2-01	Lack Of Authentication/Authorisation On RPC Services		
Asset	`provers/crates/prover-engine/src/lib.rs, crates/agglayer-grpc-api/src/lib.rs`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

In `prover-engine`, the RPC server `axum::Router` and metrics server do not appear to implement any authentication or authorisation mechanisms.

Similarly, in `crates/agglayer-grpc-api/src/lib.rs` all gRPC services are also directly mounted onto the `axum::Router` without any form of authentication, authorization or rate limiting.

If these services are exposed on a network outside of `localhost`, unauthorised users could access and interact with exposed RPC endpoints, their functionality and obtain metrics data.

Additionally, as gRPC reflection is also enabled, all registered services and methods are exposed, which could aid an attacker in reconnaissance of the API surface and crafting further attacks.

Recommendations

Add authentication (e.g., API keys, JWT) and restrict access to specific IP ranges or interfaces.

Disable gRPC reflection in production environments.

AGLO3.2-02	Network Tasks Can Drop Certificates If Channel Is Full		
Asset	`agglayer/crates/agglayer-certificate-orchestrator/src/lib.rs`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When calling the `receive_certificates()` function, if the network task channel is full, the certificate is dropped with no retry or fallback mechanism. This can result in permanent data loss.

The network task channel has fixed capacity (i.e. `DEFAULT_CERTIFICATION_NOTIFICATION_CHANNEL_SIZE = 1000`). Under heavy load, if certificates arrive faster than they can be processed, the channel will fill up, which would then result in certificates being dropped and lost.

Recommendations

Consider queuing certificates in a temporary buffer to retry later. This could be achieved by utilising `pending_store` to queue certificates when `try_reserve` fails.

AGLO3.2-03	Unencrypted Communication For RPC And Metric Endpoints		
Asset	`provers/crates/prover-engine/src/lib.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The RPC and metrics services bind to their respective `SocketAddr` interfaces using `TcpListener::bind()`, without any form of transport layer encryption or authentication:

```
let tcp_listener = prover_runtime.block_on(TcpListener::bind(addr))?;
```

If these services are exposed on a network outside of `localhost`, any data transmitted over the network (e.g., RPC requests or metrics) could be intercepted or tampered with. The risk is amplified by the presence of the gRPC reflection and health endpoints, which may expose service names and potentially encourage further reconnaissance by a malicious actor.

Recommendations

Integrate TLS support using libraries such as `rustls` or `tokio-rustls` with `axum` and `tonic` to encrypt communications.

AGLO3.2-04	Blind Reproving Of Proven Certificates Without State Recovery Or Consistency Checks		
Asset	`agglayer/crates/agglayer-certificate-orchestrator/src/network_task.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `make_progress()` function reprocesses `Proven` certificates as `Pending` without validating existing proofs or attempting recovery, risking inconsistency and unnecessary reproving.

This occurs because the `make_progress()` function handles certificates marked as `Proven` by reverting them to `Pending` and reprocessing them through the certification pipeline via `handle_pending()`. This occurs without verifying the existing proof's validity, attempting to recover the `new_state`, or ensuring consistency between the original and newly generated proofs.

The current design assumes that a `Proven` status with a missing `new_state` necessitates reproving, without exploring recovery options or validating the stored proof.

Upon encountering `CertificateStatus::Proven`, `make_progress()` logs a warning about the missing `new_state`, transitions the certificate to `Pending`, discards the existing proof from `pending_store` and invokes `handle_pending()` to reprove it, however:

- No check is performed to determine if the proof in `pending_store` is valid, or if `new_state` can be recovered, potentially discarding a usable proof
- The new proof generated by `handle_pending()` is not compared to the original, leaving potential inconsistencies undetected
- The existing proof is removed before reproving succeeds, eliminating the option to recover or compare it if reproving fails
- Reproving, a resource intensive operation, is triggered unnecessarily if the original proof was valid

Recommendations

Consider implementing the following:

- Before reproving, validate the existing proof in `pending_store` and attempt to recover `new_state`
- Retain the original proof until reproving completes successfully and consistency is confirmed
- Compare the new proof with the original to ensure they yield the same state or outcome, preventing silent inconsistencies

AGLO3.2-05	Missing Recovery For Failed Network Tasks		
Asset	`agglayer/crates/agglayer-certificate-orchestrator/src/lib.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `poll()` function when a `NetworkTask` fails (i.e. returns `Err`), the orchestrator logs the error, but does not remove the associated sender from `spawned_network_tasks`.

This means that the orchestrator incorrectly assumes the task is still alive, no new task can be spawned for that `NetworkId` and all future certificate pushes to that sender will likely fail silently or drop certificates.

Recommendations

Remove the `NetworkId` from `spawned_network_tasks` on both `Ok` and `Err` paths.

AGLO3.2-06	Unchecked Panic With `unwrap()` And `expect()`		
Asset	`provers/crates/aggchain-proof-program/src/main.rs`, `agglayer-storage/benches/latest_certificate_bench.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the proof system, multiple instances of the `unwrap()` method were observed. Particularly, on line [9] when verifying the Aggchain proof inputs. This could lead to the proving program crashing due to an unhandled panic:

```
let aggchain_proof_public_values = aggchain_witness.verify_aggchain_inputs().unwrap();
```

While `unwrap()` is a convenient way of handling `Option` and `Result` types, this can lead to unhandled panics if called on a `None` value or an `Err` variant without sufficient validation. Multiple Aggchain inputs verified inside `verify_aggchain_inputs()` can trigger an error to propagate upwards.

As some of these errors are expected in the event that the proof parameters are invalid or onchain calls fail, this outcome should be expected and protected against.

Similarly, in `crates/agglayer-storage/benches/latest_certificate_bench.rs` the code uses `unwrap()` and `expect()` extensively, which may also result in unexpected crashes on failure.

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.

AGLO3.2-07	Use of `unwrap()` with gRPC reflection server can result in panic		
Asset	`provers/crates/prover-engine/src/lib.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Both gRPC reflection servers set up by the prover engine crate make use of `unwrap()` while being built. This could cause a panic causing the prover to terminate unexpectedly:

```
let reflection_v1 = reflection_v1.build_v1().unwrap();
let reflection_v1alpha = reflection_v1alpha.build_v1alpha().unwrap();
```

However, as both builds are using default build settings, this is unlikely to happen in practice.

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.

AGLO3.2-08	`on_proven_certificate()` Continues On Partial Error		
Asset	`agglayer/crates/agglayer-certificate-orchestrator/network_task.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `on_proven_certificate()` function, the first two steps `set_latest_proven_certificate_per_network()` and `update_certificate_header_status()` log errors but proceed regardless of failure, which could result in setting `pending_state` and attempting settlement, even if these updates fail:

```
if let Err(error) = self.pending_store.set_latest_proven_certificate_per_network(...) {
    error!(...);
}
if let Err(error) = self.state_store.update_certificate_header_status(...) {
    error!(...);
}
```

Recommendations

Propagate errors from storage updates. Only set `pending_state` and attempt settlement if prior steps succeeded.

AGLO3.2-09	`insert_certificate_header()` Has No Conflict Detection For Settled Certificates		
Asset	`crates/agglayer-storage/src/stores/state/mod.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

If two different certificates are inserted for the same (`network_id` , `height`), due to an L1 race or bug in L2 consensus logic, the second one will overwrite the `CertificatePerNetworkColumn` key silently.

Recommendations

Add an explicit check for certificate conflict during insert, as per `TODO` comment on line [**168**]:

```
// TODO: Check certificate conflict during insert (if conflict it's too late)
```

AGLO3.2-10	No Panic Handling On Orchestrator Or RPC Tasks		
Asset	`crates/agglayer-node/src/node.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Neither the `Node::start()` nor `Node::await_shutdown()` functions explicitly handle potential panics in these tasks.

If a panic occurs in either the orchestrator or an RPC server task, it could cause the task to terminate silently, unwind the entire async task and terminate the node silently.

Recommendations

Wrap the orchestrator and RPC server logic in `catch_unwind` to catch panics and convert them into errors that can be logged and attempt recovery, if possible.

AGLO3.2-11	Potential Resource Leak On Panic In `LocalExecutor`		
Asset	`provers/crates/prover-executor/src/lib.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is a potential resource leak on panic in the `LocalExecutor` implementation due to how `spawn_blocking` is used.

If a panic occurs inside the blocking thread (e.g., during `prover.prove`, `run`, or `verify`), it will not be caught within the thread, causing the `JoinHandle` returned by `spawn_blocking` to be dropped without propagating meaningful context or ensuring proper clean up.

This could lead to resource leaks or incomplete error handling.

Recommendations

Wrap the blocking task body in `std::panic::catch_unwind` inside `spawn_blocking` to catch panics and convert them into a meaningful error.

AGLO3.2-12	Potential Resource Leak On Panic Before Shutdown		
Asset	`provers/crates/prover-engine/src/lib.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `start()` method spawns tasks (`prover_handle` and `metrics_handle`), but does not explicitly ensure clean up if a panic occurs before shutdown, which could lead to resource leaks.

If a panic occurs at any point after the tasks have been spawned, but before the `tokio::select!` shutdown block is reached, the runtime will continue running with orphaned background tasks and no coordinated shutdown path.

Recommendations

Consider splitting `start()` into `build()` and `run()` , so that all setup (e.g. TCP binding, reflection service build, etc.) is in a separate `build()` phase that can fail safely before tasks are spawned. Then spawn tasks only after all preparation completes successfully.

AGLO3.2-13	`write_batch()` Skips `default_write_options`		
Asset	`crates/agglayer-storage/src/storage/mod.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In `crates/agglayer-storage/src/storage/mod.rs` the code is explicitly setting `sync = true` on `default_write_options` in `open_cf`, but in `write_batch()` the `write()` function is used, which will use RocksDB's default behaviour:

```
pub fn write_batch(&self, batch: WriteBatch) -> Result<(), DBError> {
    self.rocksdb.write(batch)?;     // @audit no write options used
    Ok(())
}
```

This bypasses the `default_write_options`, and therefore, previously set sync durability is not guaranteed.

Recommendations

Use `write_opt()` function instead, e.g.:

```
self.rocksdb.write_opt(batch, &self.default_write_options)?;
```

AGLO3.2-14	Partial Error Handling In `create_new_backup()`		
Asset	`crates/agglayer-storage/src/storage/backup/mod.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

In `crates/agglayer-storage/src/storage/backup/mod.rs` the `create_new_backup()` function logs errors for individual backup operations (state, pending, epoch), but returns `Ok(())` regardless:

```
if let Err(error) = self.state_engine.create_new_backup_flush(...) {
    error!("Failed to create backup for state db: {:?}", error);
}

// ...

info!("Backup successfully created");

Ok(())
```

This results in callers not being informed of failures, potentially assuming a successful backup.

Recommendations

Aggregate errors and fail if any operation fails.

AGLO3.2-15	Unimplemented GPU Prover Type
Asset	`prover/crates/prover-executor/src/lib.rs`
Status	Open
Rating	Informational

Description

`ProverType::GpuProver` uses `todo!()`, which will panic if used, crashing the application unexpectedly.

Recommendations

Return an error instead, e.g.:

```
ProverType::GpuProver(_) => Err(Error::UnsupportedProver("GPU prover not implemented".to_string()))?,
```

AGLO3.2-16	Fallback Mechanism Cloning Overhead
Asset	`provers/crates/prover-executor/src/lib.rs`
Status	**Open**
Rating	Informational

Description

In `Executor::call`, the fallback service is cloned on every request (`let fallback = self.fallback.clone();`), even if the primary succeeds.

Unnecessary cloning adds overhead, especially since `BoxCloneService` involves dynamic allocation.

Recommendations

Modify implementation to defer the cloning of the fallback until it is actually needed, after the primary fails.

AGLO3.2-17	Fallback Service Readiness Not Polled In Executor::poll_ready
Asset	provers/crates/prover-executor/src/lib.rs
Status	Open
Rating	Informational

Description

In the `Executor` implementation, the `poll_ready()` method checks only the primary service's readiness, but not the fallback's:

```
fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll> {
    self.primary.poll_ready(cx)
}
```

If the primary fails and the fallback is not ready, the call method will still attempt to use it, potentially leading to delays or errors.

Recommendations

Check both services' readiness if fallback is present.

AGLO3.2-18	No Resource Cleanup At Shutdown
Asset	`provers/crates/prover-engine/src/lib.rs`
Status	Open
Rating	Informational

Description

Shutdown relies on `CancellationToken` and waiting for handles, but there's no timeout or explicit resource cleanup.

Recommendations

Add a configurable shutdown timeout and ensure all resources (e.g., TCP listeners) are explicitly closed:

```
cancellation_token.cancel();
tokio::time::timeout(Duration::from_secs(5), prover_handle).await??;
tokio::time::timeout(Duration::from_secs(5), metrics_handle).await??;
```

AGLO3.2-19	Hardcoded Default Socket Addresses
Asset	`provers/crates/prover-engine/src/lib.rs`
Status	Open
Rating	Informational

Description

The code uses hardcoded default socket addresses (`{[]::1{]}:10000` for RPC and `{[]::1{]}:10001` for telemetry) if none are provided via `set_rpc_socket_addr()` or `set_metric_socket_addr()`.

```
let addr = self.rpc_socket_addr.take().unwrap_or_else(|| {
            "[::1]:10000"
                .parse()
                .expect("Unable to parse the RPC socket address")
        });
        let telemetry_addr = self.metric_socket_addr.take().unwrap_or_else(|| {
            "[::1]:10001"
                .parse()
                .expect("Unable to parse the telemetry socket address")
        });
```

Hardcoding addresses could lead to port conflicts in production environments or unexpected behaviour if the application binds to an unintended interface.

Recommendations

Use configurable defaults (e.g., via environment variables or a config file) instead of hardcoded values.

AGLO3.2-20	`LocalNetworkStateData` State Risks Incorrect State Recorded On Error Conditions
Asset	`agglayer/crates/agglayer-certificate-orchestrator/certifier.rs`
Status	**Open**
Rating	Informational

Description

In `crates/agglayer-certificate-orchestrator/src/certifier.rs`, the `witness_execution()` function takes a mutable reference to `LocalNetworkStateData` which is modified in place, risking partial updates or state corruption if an error occurs during execution.

If an implementor panics or fails during mutation, the state could be left inconsistent, potentially allowing attacks such as partial state updates.

Recommendations

Refactor the code to operate on a cloned or owned version of state.

AGLO3.2-21	Task Spawning Without Limits
Asset	`agglayer/crates/agglayer-certificate-orchestrator/src/lib.rs`
Status	**Open**
Rating	Informational

Description

In `crates/agglayer-certificate-orchestrator/src/lib.rs`, the function `spawn_network_task()` creates a new task for each network ID, without a cap on the number of tasks.

A large number of networks could spawn excessive tasks, exhausting system resources.

Recommendations

Consider introducing a configurable limit on concurrent network tasks.

AGLO3.2-22	Miscellaneous General Comments
Asset	*
Status	Open
Rating	Informational

Description

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

1. **Health Status Set Before Services Start**

   Related Asset(s): provers/crates/prover-engine/src/lib.rs

   The health status of services is set to Serving before the RPC server is fully operational (i.e., before `axum::serve` is running).

   This could mislead clients into thinking the service is ready when it is still initialising, potentially causing connection errors.

   Defer setting the health status until after the server is bound and running.

2. **Blocking Calls Inside Tokio Runtime Context**

   Related Asset(s): provers/crates/prover-engine/src/lib.rs

   `Runtime::block_on()` inside the `start()` function could block the runtime unnecessarily and lead to surprising behaviour if the function is ever called from an async context.

   Consider replacing with `await` and refactoring `start()` to be fully async function. Otherwise, clearly document that `start()` must only be called synchronously.

3. **Redundant Reflection Service Registration**

   Related Asset(s): provers/crates/prover-engine/src/lib.rs

   There are two `.fold()` loops over `self.reflection` with the same logic. Once on line [**155**] and again on line [**168**].

   This redundancy is unnecessary and increases computation time without adding value.

   Remove the second fold operation.

4. **Commented Out Code**

   Related Asset(s): provers/crates/prover-engine/src/lib.rs

   Some code is commented out and so not functional. See lines [**238-239**]

   ```
   // prover_runtime.shutdown_timeout(config.shutdown.runtime_timeout);
   // metrics_runtime.shutdown_timeout(config.shutdown.runtime_timeout);
   ```

   Commented out code should be reviewed and uncommented if intended to be used or deleted if redundant.

5. **Address TODOs**

   Related Asset(s): *

   Address `TODO` comments throughout the codebase.

   For example, in `crates/agglayer-storage/src/stores/state/mod.rs` line [**168**]:

   ```
   // TODO: Check certificate conflict during insert (if conflict it's too late)
   ```

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.

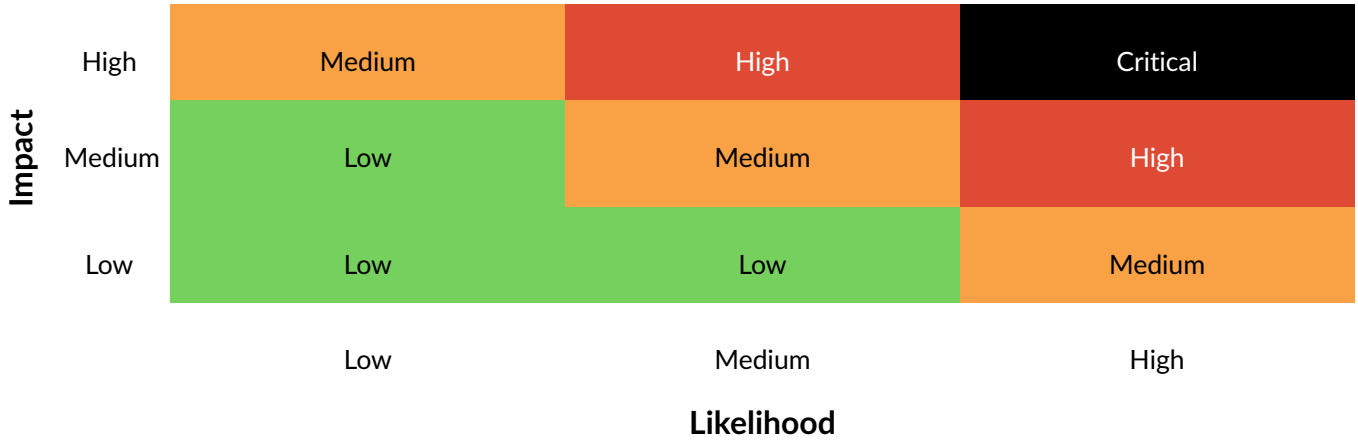

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.