



Blockchain Protocol Security Analysis Report

Customer: Somnia

Date: 04/07/2025



We express our gratitude to the Somnia team for the collaborative engagement that enabled the execution of this Blockchain Protocol Security Assessment.

Somnia is a high-performance Layer 1 blockchain designed for EVM compatibility. It utilizes a Practical Byzantine Fault Tolerance (PBFT) consensus mechanism and features a unique architecture that separates data dissemination from consensus.

Document

Name	Blockchain Protocol Review and Security Analysis Report for Somnia
Audited By	Ali Ashar, Stephen Ajayi
Approved By	Nino Lipartia
Website	https://somnia.network/
Changelog	17/06/2025 - Preliminary Report
	04/07/2025 - Final Report
Platform	Somnia Network
Language	C++
Tags	L1, Consensus, PBFT
Methodology	https://hackenio.cc/blockchain_methodology

Review

Scope

Repository	https://github.com/metaversal-foundation/somnia-audit-repo/tree/main
Commit	0af8c04569b626dd5c0cace903998b9ec06ab5a6

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

16	6	10	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	1
Medium	4
Low	7

Vulnerability	Severity
F-2025-10995 - State Divergence via Flawed Vote Aggregation in View Change	Critical
F-2025-10890 - State Poisoning via Unverified Fetched Request Batches	High
F-2025-10817 - Delayed Signature Verification in Checkpoint Message Processing	Medium
F-2025-10970 - Unbounded growth of request_timeouts leading to CPU & memory DoS	Medium
F-2025-10987 - Unbounded in_flight_fetches Growth in SkipToRequestBatch	Medium
F-2025-10998 - Unbounded growth of transient_views leading to memory exhaustion	Medium
F-2025-10425 - PBFT Parameters Lack Bounds Validation Allowing DoS via Misconfiguration	Low
F-2025-10758 - Lack of Rate-Limiting on Signature Verification	Low
F-2025-10784 - Clock Skew Can Induce Unnecessary Potential False View-Changes	Low
F-2025-10837 - Configuration Risk Allowing Permanent Liveness Failure	Low
F-2025-10873 - Excessive CPU in Tick() Loop	Low
F-2025-10903 - Dangling PbftNodeContext Reference in PbftManagerFactory	Low
F-2025-10999 - Uninitialized collaborator pointers in PbftViewManager	Low
F-2025-10528 - Missing Committee Membership Validation	Info
F-2025-10897 - Implicit Copyability of PbftCheckpointManager	Info
F-2025-10976 - Unbounded Growth of pending_sent_requests leading to CPU & Memory DoS	Info

Documentation quality

- A detailed technical design document for the PBFT implementation was provided, outlining the architecture and deviations from the original academic paper.
- The repository's `README.md` file is minimal and lacks crucial information for new developers, such as a clear project overview, detailed build instructions, and guidance on running the test suites.
- While the code is well-structured with inline comments, the project would benefit from a more formal developer guide for onboarding.

Code quality

- The C++ codebase is modern and built from scratch, with a modular architecture that separates concerns into different manager classes.
- Static analysis revealed several areas for improvement, including initialization-order bugs, self-initialization, and inconsistent coding styles.
- The project uses the Bazel build system, providing a configured and reproducible development environment.

Architecture quality

- **Scalability:** The architecture separates high-throughput data dissemination from low-latency PBFT consensus, a design intended to support high performance at the execution layer.
- **Decentralization:** The system uses a delegated Proof-of-Stake model, where a committee of nodes is chosen each epoch to participate in consensus. Nodes not on the committee act as non-voting replicas.
- **Performance:** The PBFT implementation is optimized for low latency and includes a "fast path" commitment to reduce network round trips.
- **Interoperability:** The system is designed for EVM compatibility, with the consensus layer ordering opaque data that the execution layer interprets as Ethereum transactions.
- **Resilience:** Vulnerabilities affecting consensus safety and resource management were identified, including flawed vote aggregation, state poisoning vectors, and unbounded internal map growth. These gaps indicate areas requiring protocol-level hardening to ensure safety under Byzantine conditions.

Test coverage

- The project includes a comprehensive testing framework, featuring unit tests, integration tests on a 4-node network, a fuzzer for liveness testing, and a dedicated Byzantine test suite.
- While happy-path and some failure scenarios are covered, this audit identified critical negative cases (e.g., resource exhaustion, state divergence attacks) that were not previously covered by tests.
- The Byzantine test suite is a strong foundation but needs to be expanded to cover the attack vectors identified in this report.

Table of Contents

System Overview	6
Threat Model	6
Risks	7
Findings	8
Vulnerability Details	8
F-2025-10995 - State Divergence Via Flawed Vote Aggregation In View Change - Critical	8
F-2025-10890 - State Poisoning Via Unverified Fetched Request Batches - High	13
F-2025-10817 - Delayed Signature Verification In Checkpoint Message Processing - Medium	16
F-2025-10970 - Unbounded Growth Of Request_timeouts Leading To CPU & Memory DoS - Medium	20
F-2025-10987 - Unbounded In_flight_fetches Growth In SkipToRequestBatch - Medium	22
F-2025-10998 - Unbounded Growth Of Transient_views Leading To Memory Exhaustion - Medium	24
F-2025-10425 - PBFT Parameters Lack Bounds Validation Allowing DoS Via Misconfiguration - Low	26
F-2025-10758 - Lack Of Rate-Limiting On Signature Verification - Low	28
F-2025-10784 - Clock Skew Can Induce Unnecessary Potential False View-Changes - Low	31
F-2025-10837 - Configuration Risk Allowing Permanent Liveness Failure - Low	35
F-2025-10873 - Excessive CPU In Tick() Loop - Low	38
Compile And Run	40
F-2025-10999 - Uninitialized Collaborator Pointers In PbftViewManager - Low	44
F-2025-10528 - Missing Committee Membership Validation - Info	46
F-2025-10897 - Implicit Copyability Of PbftCheckpointManager - Info	48
F-2025-10976 - Unbounded Growth Of Pending_sent_requests Leading To CPU & Memory DoS - Info	50
Observation Details	52
Disclaimers	53
Hacken Disclaimer	53
Technical Disclaimer	53
Appendix 1. Severity Definitions	54
Appendix 2. Scope	55
Components In Scope	55

System Overview

Somnia is a high-performance Layer 1 Proof-of-Stake blockchain designed for EVM compatibility. Its architecture separates core blockchain functions into two distinct layers:

- **Data Dissemination Layer:** Individual validator nodes produce their own "data chains" and disseminate new blocks to their peers. A proof-of-availability mechanism, based on collecting signatures from peers, is used to validate the advancement of these chains.
- **Consensus Layer:** This layer uses a Practical Byzantine Fault Tolerance (PBFT) consensus protocol. Instead of ordering individual transactions, the PBFT network's role is to agree on the order of the latest "tip" hashes from the various data chains. This design allows the consensus mechanism to be low-throughput and optimized for low latency, while the data layer handles high-throughput transaction data.

Consensus is performed by a "committee" of validator nodes chosen each epoch based on their staked cryptocurrency.

Threat Model

This audit assumes a partially synchronous network with a validator committee of $3f+1$ voting power, tolerating up to f Byzantine participants. Threats analyzed include:

- Consensus safety failures from equivocation or inconsistent PrePrepare propagation
- Liveness failures from DoS on timeout loops or invalid message floods
- Resource exhaustion through unbounded maps or bypassed signature checks
- Edge-case misconfigurations due to unsafe default parameters or flags

Risks

- The protocol's security and liveness depend on a "committee" of validator nodes chosen each epoch by an on-chain smart contract. This design introduces a potential point of centralization. Any vulnerability, governance weakness, or exploit in the committee selection contract could allow a malicious actor to influence the committee's composition, which could compromise the entire consensus mechanism. The security of this external smart contract is therefore paramount to the security of the L1 protocol itself.
- **Reliance on Operator Correctness:** Several potential issues, such as those related to clock skew and dangerous debug parameters, are mitigated by the expectation that node operators will use server-grade hardware with proper time synchronization and will not misconfigure their nodes. This places a significant trust assumption on the technical competence and diligence of all operators, as simple configuration errors could lead to individual node failures that degrade the network's overall fault tolerance.

Vulnerability Details

[F-2025-10995](#) - State Divergence via Flawed Vote Aggregation in View Change - Critical

Description:

The core safety of the PBFT protocol during a view change depends on correctly identifying which `RequestBatch` was "locked" by a quorum of honest nodes in a previous view. This locked batch must be preserved and re-proposed in the new view to maintain safety.

The function `PbftViewManager::GetLockedRequestBatchesForNewView` performs this logic by analyzing `LeaderViewChangeMessages` received from peers to determine which proposals have sufficient support. However, the current vote aggregation mechanism is flawed: it aggregates votes using only the `(sequence_number, view_number)` tuple, without considering the `request_batch_hash`.

As a result, if a Byzantine leader proposes different `RequestBatches` for the same sequence and view to different subsets of validators, the vote counter will sum these as if they supported the same batch, incorrectly reaching the $f+1$ vote threshold and "locking" a batch that was never legitimately agreed upon.

Affected Code

File: `pbft_view_manager.cc`

The vulnerability lies in the data structure used to count votes. It lacks the necessary granularity to distinguish between different proposals for the same slot.

```
// This map is keyed only by sequence number and view number.
// It does not account for the request batch hash.
unordered_map<SequenceNumber, unordered_map<ViewNumber, std::uint64_t>
>> votes_per_proposal;

// ...

for (const auto& view_change_message : view_change_messages) {
    // ...
    for (const auto& [sequence_number, proposed_round] :
        view_change_message.preprepared_rounds_above_checkpointed) {
        // ...

        // This line incorrectly increments the vote count for the (view, s
        // equence)
        // pair, regardless of whether the proposed_round.request_batch is
        // the
        // same as one seen from another peer.
        auto& votes = votes_per_proposal[sequence_number][proposed_round.vi
```

```
ew_number];
votes += epoch_data.GetNumVotes(*view_change_sender);

if (votes >= kFPlusOneVotes) {
// This block can be entered with votes for conflicting batches.
// ...
}
}
}
```

Impact

This logic flaw enables a Byzantine proposer to compromise PBFT safety and cause honest nodes to commit different blocks for the same sequence number. This is a **catastrophic consensus failure** that can result in **irreversible blockchain forks**, undermining the integrity of the canonical chain.

Attack Scenario

1. **Split Proposal:** A Byzantine leader in view V sends `PrePrepare(S, Batch_A)` to half the honest validators, and `PrePrepare(S, Batch_B)` to the other half.
2. **View Change:** A view change to V+1 is triggered. Honest nodes include the pre-prepared batch they saw in their `LeaderViewChangeMessage`.
3. **Flawed Vote Counting:** The new leader aggregates these votes based on `(S, V)`, ignoring the batch hash, and falsely concludes that a single batch has enough support.
4. **Locking Divergent State:** The new leader locks in one batch (e.g., Batch B), but a different honest node could just as easily lock in Batch A.
5. **Safety Failure:** The network now has **two conflicting, locked states for the same sequence number**, violating the core safety property of PBFT.

Assets:

- somnia/pbft/pbft_view_manager.cc
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 4/5

Severity: Critical

Recommendations

Remediation:

To remediate this, the vote aggregation logic must be made more granular by including the `request_batch_hash` in the vote key. This ensures that votes are only accumulated for identical proposals.

1. Modify the Vote Counting Data Structure: The map key must include the hash of the request batch.

```
// Inside the loop processing prepared_rounds_above_checkpointed
// --- RECOMMENDED CHANGE ---
const ProposalKey proposal_hash = proposed_round.request_batch.request_batch_hash;
auto& votes = votes_per_proposal[sequence_number][proposed_round.view_number][proposal_hash];
votes += epoch_data.GetNumVotes(*view_change_sender);

if (votes >= kFPlusOneVotes) {
// This block is now only reached if a single, specific batch has f
+1 votes.
// ...
}
```

2. Update the Vote Aggregation Logic: Use the `request_batch_hash` when incrementing the vote count.

```
// --- RECOMMENDED CHANGE ---
// The inner map should be keyed by the request batch hash.
using ProposalKey = HashableHash;
UnorderedMap<SequenceNumber, UnorderedMap<ViewNumber, UnorderedMap<
ProposalKey, std::uint64_t>>> votes_per_proposal;
```

This change ensures that only votes for the exact same proposal are aggregated, closing the state divergence vulnerability.

Resolution:

The finding is fixed in the commit [fb5d24a](#).

The vote aggregation logic in the

`PbftViewManager::GetLockedRequestBatchesForNewView` function was updated.

The key used for counting votes now includes the `request_batch_hash`, ensuring that only votes for identical proposals are aggregated.

Evidences

POC: State Divergence Test via Conflicting ViewChange Aggregation

Reproduce:

1. Create a file named `poc_state_divergence.cc` and include the binary in the BUILD

```
#include "somnia/pbft/pbft_view_manager.h"
#include "somnia/epoch/epoch_manager.h"
#include "somnia/test/test_user.h"
```

```

#include "somnia/lib/common.h"
#include <iostream>
#include <vector>
using namespace somnia;
using namespace somnia::pbft;
// --- TESTING WRAPPER FOR PRIVATE FUNCTION ACCESS ---
// Grants access to the internal view change logic that determines
// what gets locked during a new view
struct PbftViewManagerTestWrapper : public PbftViewManager {
static void CallGetLockedRequestBatchesForNewView(
const PbftNetworkSetup& network_setup,
const EpochData& epoch_data,
const PbftInterface& interface,
const std::vector<LeaderViewChangeMessage>& view_change_messages,
std::vector<ViewChangeRoundCertificateData>& rounds_to_commit,
std::vector<std::pair<SequenceNumber, Optional<RequestBatch>>>& rounds_to_prepare
) {
GetLockedRequestBatchesForNewView(
network_setup,
epoch_data,
interface,
view_change_messages,
rounds_to_commit,
rounds_to_prepare
);
}
};
// --- BATCH CREATION HELPERS ---
// Creates a fake request batch with unique payload and hash
RequestBatch CreateDummyBatch(SequenceNumber seq, int id) {
RequestBatch batch;
batch.network_descriptor = {100, {}};
batch.sequence_number = seq;
std::string data_str = "batch_data_" + std::to_string(id);
std::vector<uint8_t> data(data_str.begin(), data_str.end());
RequestPayload payload;
payload.data = data;
payload.payload_hash = payload.CalculatePayloadHash();
batch.request_payloads.push_back(payload);
batch.proposer_address = NonRandomAddress(id);
batch.timestamp = 0;
batch.sequence_commitment = HashFromInt(seq);
batch.request_batch_hash = batch.CalculateHash(); //

```

[See more](#)

Results:

```

--- Running Standalone POC for State Divergence (F-2025-10995) ---
[2025-06-16 18:08:38.116] [poc_state_diver] [info] Setting peer 0 t
o 0xd1d8a091d3644d1a8ee6b995939bf85c41215b6f with 826461203 votes i
n epoch 0.
[2025-06-16 18:08:38.116] [poc_state_diver] [info] Setting peer 1 t
o 0x0b5a1df83bf4430a8433bb8e375bb5b4e02411b5 with 720486850 votes i
n epoch 0.
[2025-06-16 18:08:38.116] [poc_state_diver] [info] Setting peer 2 t
o 0x9e3794a4c84a63399253af258183429896bbc81d with 585916409 votes i
n epoch 0.
[2025-06-16 18:08:38.116] [poc_state_diver] [info] Setting peer 3 t
o 0x6b626340e0cb9d5f9fc0adb86778e846c44a4020 with 867135539 votes i
n epoch 0.
Setup: 4-node committee (f=1). Quorum for locking a pre-prepare is
f+1 = 2 votes.
Scenario: Byzantine leader sends conflicting batches for sequence 0
:
- Node 1 sees Batch A (Hash: 0xa9cff9a02a3f4dff297b94f904f575820de2
6db69040dad7e6b3e273df57d469)
- Node 2 sees Batch B (Hash: 0x4f4075d053adc61b5b33c2bdc9c961853468
0522e6d300b000b06990ad02dd21)
Simulating view change: New leader receives view change messages fr
om Nodes 1 & 2.

!!! VULNERABILITY CONFIRMED !!!
A batch for sequence 0 was locked, despite no single batch having f
+1 votes.
The flawed logic combined votes for different batches.
Locked Batch Hash: 0x4f4075d053adc61b5b33c2bdc9c9618534680522e6d300
b000b06990ad02dd21

```

POC Status: SUCCESS

[F-2025-10890](#) - State Poisoning via Unverified Fetched Request Batches - High

Description:

The node accepts fetched `RequestBatch` objects from peers and commits them into local state **without verifying their internal integrity**. Specifically, the implementation does not call `**VerifyHashes()` on each `RequestBatch` before it is accepted - relying solely on a cumulative hash built from their outer `request_batch_hash` values.

This allows a malicious peer to forge a batch with **correct external hash**, but **corrupted internal data**. The node will treat it as valid and commit it to state, violating the core PBFT safety property that all honest nodes must execute the same requests in the same order.

Impact

This is a critical safety violation. A single malicious peer can permanently corrupt the state of any syncing node. This can be exploited for targeted attacks (e.g., against an exchange's node), can cause a node's state to become irrecoverably different from the rest of the network, and breaks the guarantee of transaction finality.

Why is it vulnerable?

- The system trusts the outer `request_batch_hash` without checking if it matches the actual payload contents.
- `VerifyHashes()` — the function that checks internal data integrity — is never called before batches are committed to state.
- The logic assumes that if the cumulative hash passes, all data is valid - which is incorrect.

Logical Flaw in Code:

In `pbft_committed_request_batch_manager.cc`, the callback for `fetch_missing_requests_resource_system.RequestObject` performs a cumulative hash check. If this passes, it enters the final loop to commit the batches:

```
// from pbft_committed_request_batch_manager.cc

// ... (previous checks on sequence numbers and cumulative hash pas
s) ...

// These are the correct request batches. Add them locally.
RELEASE_ASSERT(cumulative_request_batch_hashes_per_request_batch.size() ==
response.request_batches.size() + 1);
for (std::size_t i = 0; i < response.request_batches.size(); i++) {
// CRITICAL OMISSION: No call to response.request_batches[i].Verify
Hashes() here.
committed_request_batches.emplace(
response.request_batches[i].sequence_number,
```

```
CommittedRequestBatch{response.request_batches[i],
cumulative_request_batch_hashes_per_request_batch[i + 1]}};
}
```

The code proceeds to `emplace` the received batches directly into the state without first calling the `VerifyHashes()` method on each one. This is the missing validation step.

Assets:

- `somnia/pbft/pbft_committed_request_batch_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Severity: High

Recommendations

Remediation: Apply a strict validation before committing peer-fetched batches:

```
for (const RequestBatch& batch : response.request_batches) {
  if (!batch.VerifyHashes()) {
    // Reject entire response. Optionally penalize peer.
    return false;
  }
}
```

This should be done **before** calculating cumulative hashes and **before** any state update occurs.

Resolution:

The finding is fixed in the commit [fb5d24a](#).

A validation loop was added in the `PbftCommittedRequestBatchManager`. This ensures that every fetched `RequestBatch` is now individually verified via `VerifyHashes()` before being accepted.

Evidences

Proof of Concept Test (state_poison_catchup_test.cc)

Reproduce:

The provided test case successfully demonstrates this flaw.

- It creates a "good" batch and a "forged" batch with the same outer `request_batch_hash` but different internal data.
- It proves the forged batch is internally invalid via `ASSERT_FALSE(forged.VerifyHashes())`.
- It then proves that the manager's cumulative hash logic cannot distinguish between the two via `EXPECT_EQ(legit_cum, forged_cum)`.

Reproduce the test:

- Create `state_poison_catchup_test.cc` in the `pbft_byzantine_test` folder
- Use this test code:

```
#include "gtest/gtest.h"
#include "somnia/pbft/pbft_types.h"
#include "somnia/pbft/pbft_committed_request_batch_manager.h"

using namespace somnia;
using namespace somnia::pbft;

// — Helper: Copy a valid batch and poison its payload —————
static RequestBatch ForgePoisonedBatch(const RequestBatch& original) {
    RequestBatch poisoned = original;
    poisoned.request_payloads[0].data = {'B', 'A', 'D'}; // Corrupt internal data
    // Do NOT update hashes – this simulates a forged batch
    return poisoned;
}

TEST(StatePoison, CumulativeHashAcceptsForgedBatch) {
    // 1: To create a valid batch
    RequestBatch valid;
    valid.sequence_number = 42;
    valid.request_payloads.resize(1);
    valid.request_payloads[0].data = {'O', 'K'};
    valid.request_payloads[0].payload_hash = valid.request_payloads[0].CalculatePayloadHash();
    valid.request_batch_hash = valid.CalculateHash();

    // 2: To then forge a poisoned version with the same outer hash
    RequestBatch forged = ForgePoisonedBatch(valid);
    ASSERT_FALSE(forged.VerifyHashes())
    << "Poisoned batch unexpectedly passes internal hash checks.";

    // 3: Check cumulative hash logic (used by SkipToRequestBatch)
    Hash zero{};
    Hash hash_valid = UpdateCumulativeRequestBatchHash(zero, valid.request_batch_hash);
    Hash hash_f
```

[See more](#)

Results:

The successful execution of the PoC test confirms the vulnerability:

```
[ RUN ] StatePoison.CumulativeHashAcceptsForgedBatch
[ OK ] StatePoison.CumulativeHashAcceptsForgedBatch (0 ms)
[ PASSED ] 1 test.
```

This passing result confirms that the cumulative hash check is insufficient and would allow a poisoned batch to be accepted.

[F-2025-10817](#) - Delayed Signature Verification in Checkpoint Message Processing - Medium

Description:

In `pbft_checkpoint_manager.cc`, the function

`PbftCheckpointManager::ReceivedCheckpointMessage()` accepts and stores `CheckpointMessage` instances based solely on committee membership (`epoch_data.IsInCommittee()`), deferring signature verification until quorum certificate (QC) construction in `TryBuildCheckpointQuorumCertificate()`.

This design allows an attacker (including a malicious in-committee node) to flood the system with:

- **Invalid signatures**, forcing unnecessary cryptographic operations during QC assembly.
- **Valid signatures for incorrect hashes**, bloating memory with unusable messages.

Because messages are stored before validation, it enables **resource exhaustion** attacks, degrading node performance and potentially disrupting checkpoint synchronization.

Why It Matters

Castro & Liskov's original PBFT paper explicitly requires that every incoming authenticated message be signature-verified before being logged or counted (e.g., *"A backup accepts a pre-prepare message provided: ... the signatures in the request and the pre-prepare message are correct ..."*). Delaying this check violates the core safety/authentication invariant of the protocol.

Moreover, Autobahn's specification also emphasizes that *"we expect that all signatures and quorums are validated, and that external data validity predicates are enforced"*.

Affected Code

```
void PbftCheckpointManager::ReceivedCheckpointMessage(CheckpointMessage message) {  
    ...  
    if (!epoch_data.IsInCommittee(message.signature.signer)) {  
        return;  
    }  
    if (checkpoint.received_checkpoint_messages.count(message.signature.signer)) {  
        return;  
    }  
    // Missing: Early signature verification  
    checkpoint.received_checkpoint_messages.emplace(message.signature.signer, message);  
}
```

Why It's Vulnerable

- Signature (`PartialSignature`) is only verified during QC construction, not upon receipt.
- Unvalidated messages are stored in `received_checkpoint_messages`, wasting memory.
- Invalid messages force redundant verification later in `TryBuildCheckpointQuorumCertificate()`.

Impact

- **Performance Degradation:** Wasted CPU cycles and memory by processing invalid messages.
- **DoS Risk:** Flooding can delay legitimate QC assembly.
- **Protocol Compliance:** Liveness is weakened even if PBFT safety is preserved.

Assets:

- `somnia/pbft/pbft_byzantine_test/pbft_checkpoint_manager.cc` [<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: Immediately verify each `CheckpointMessage`'s partial signature before storing it. Drop any message whose signature fails.

Sample Code Fix:

The sample fix below rejects invalid signatures immediately, preventing storage and later CPU work.

```
// After checking committee membership:
// Verify the partial signature now, before storing.
if (!message.signature.Verify(message.GetMessageHash(), epoch_data)
) {
    spdlog::warn("Invalid checkpoint signature from {}", message.signat
ure.signer);
    return;
}
```

```
if (checkpoint.received_checkpoint_messages.count(message.signature
.signer)) {
return;
}
checkpoint.received_checkpoint_messages.emplace(
message.signature.signer, message);
```

Resolution:

The finding is fixed in the commit [fb5d24a](#).

The `ReceivedCheckpointMessage` function in `PbftCheckpointManager` was updated. It now performs an immediate signature verification on a new `CheckpointMessage` if an entry for that sender already exists.

Evidences

Proof of Concept (PoC):

Reproduce:

Here's a minimal standalone PoC, you'll see that even "bad" signatures get accepted and stored, leading to unbounded growth.

create a file `missing_sig_verify.cpp`

```
// missing_sig_verify.cpp
//
// Proof-of-Concept for: Delayed Signature Verification in Checkpoint
// Message Processing
#include <iostream>
#include <map>
#include <string>
#include <cassert>

using Address = uint64_t;

// Simulated CheckpointMessage structure
struct CheckpointMessage {
Address signer;
std::string signature_bytes;
};

// Simulated committee check
bool IsInCommittee(Address) {
return true; // Assume all signers are committee members for this P
oC
}

// Simulated signature verification function
bool VerifySignature(const CheckpointMessage& msg) {
return msg.signature_bytes == "GOOD_SIG";
}

// Vulnerable manager: stores messages without verifying signature
class VulnerableCheckpointManager {
std::map<Address, CheckpointMessage> received_checkpoint_messages;
public:
void ReceivedCheckpointMessage(const CheckpointMessage& msg) {
if (!IsInCommittee(msg.signer))
return;

if (received_checkpoint_messages.count(msg.signer))
return;

// Vulnerable: No early signature verification
received_checkpoint_messages.emplace(msg.signer, msg);
}

size_t size() const {
return received_checkpoint_messages.size();
}
```

```

    }
};

int main() {
    VulnerableCheckpointManager manager;

    const int TOTAL_MESSAGES = 1000;

    // Send 1000 messages: 500 valid, 500 invalid
    for (int i = 0; i < TOTAL_MESSAGES; ++i) {
        CheckpointMessage msg;
        msg.signer = i;
        msg.signature_bytes = (i % 2 == 0) ? "GOOD_SIG" : "BAD_SIG"; // Even-indexed are valid

        manager.ReceiveCheckpointMessage(msg);
    }

    // Expectation: if signatures were verified early, only 500 messages would be stored
    // Actual: all

```

[See more](#)

[F-2025-10970](#) - Unbounded growth of request_timeouts leading to CPU & memory DoS - Medium

Description: The `request_timeouts` map (`OrderedMap<Hash, RequestTimeout>`) is populated in `TryAddRequestTimeout()` for every incoming `RequestPayload` without any upper bound. An attacker can flood the node with unique payload hashes (valid or invalid), causing `request_timeouts` to grow unbounded. On each `Tick()`, `ProcessRequestTimeouts()` scans the entire map twice, performing hash comparisons, status checks, and potentially invoking view changes. This $O(N)$ scan on $O(N)$ entries every tick can exhaust CPU and memory.

Affected Code

```
//somnia/pbft/pbft_received_request_manager.cc
void PbftReceivedRequestManager::TryAddRequestTimeout(const Request
Payload& request_payload) {
// ...
request_timeout.expiry_time = current_time + timeout_manager->Curre
ntReceivedRequestTimeout();
request_timeouts.emplace(request_payload.payload_hash, std::move(re
quest_timeout));
}

void PbftReceivedRequestManager::ProcessRequestTimeouts() {
//...
// First pass: remove non-executable
for (const auto& [hash, req] : request_timeouts) { ... }
//...
// Second pass: detect expired, validate, maybe view change
for (const auto& [hash, req] : request_timeouts) { ... }
}
```

Impact

- **CPU Exhaustion:** $O(N)$ scans every tick (e.g. every 100 ms) over thousands or millions of entries burns CPU.
- **Memory Exhaustion:** Unbounded map growth can consume all available memory (DoS).
- **Consensus Stall:** High resource pressure can delay or halt consensus progress.

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation: Impose a configurable cap on `request_timeouts` size (e.g. `max_pending_requests`).

```
constexpr size_t kMaxRequestTimeouts = 10'000; // Configurable
void TryAddRequestTimeout(const RequestPayload& request_payload) {
    if (request_timeouts.size() >= kMaxRequestTimeouts) {
        // Evict oldest entry (or random)
        request_timeouts.erase(request_timeouts.begin());
    }
    // Add new entry
    request_timeouts.emplace(...);
}
```

Reject invalid requests before adding to `request_timeouts`:

```
if (!interface.ValidateRequest(...)) {
    return; // Skip unvalidatable requests
}
```

Optionally, rate-limit incoming `ReceivedRequest()` calls per sender or per view.

Resolution: The finding is fixed in the commit [fb5d24a](#).

The `OnViewChanged()` function was implemented in the `PbftReceivedRequestManager`. This function clears the `request_timeouts` map entirely upon every view change.

[F-2025-10987](#) - Unbounded in_flight_fetches Growth in SkipToRequestBatch - Medium

Description:

`SkipToRequestBatch` records each fetch attempt in the `in_flight_fetches` map without any cap on its size or the sequence-range requested. A malicious or misconfigured peer can repeatedly trigger large skips, causing `in_flight_fetches` to grow indefinitely.

Affected Code

```
// pbft_committed_request_batch_manager.h
OrderedMap<SequenceNumber, InFlightFetch> in_flight_fetches;

// pbft_committed_request_batch_manager.cc
void PbftCommittedRequestBatchManager::SkipToRequestBatch(...) {
  InFlightFetch fetch{ from = next_seq, to = sequence_number, ... };
  in_flight_fetches.emplace(fetch.from_sequence_number, fetch);
  // ... issue fetch ...
}
```

Impact

- **Memory exhaustion:** map grows without bound → OOM.
- **CPU burn:** cleanup logic may scan large `in_flight_fetches`.
- **Consensus stall:** node overwhelmed, drops behind peers.

Assets:

- `somnia/pbft/pbft_committed_request_batch_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation:

- Enforce a maximum fetch range (e.g. `sequence_number - next_seq ≤ max_fetch_range`).
- Cap the number of concurrent fetches (e.g. `in_flight_fetches.size() ≤ max_concurrent_fetches`).

- Evict or fail stale entries after timeout.

Resolution:

The finding is Accepted by the client.

The underlying `ResourceSystem` has an "always eventually succeed" semantic. It is designed to never fail or time out and will hold a request until it is successful, at which point the entry in `in_flight_fetches` is cleared.

F-2025-10998 - Unbounded growth of transient_views leading to memory exhaustion - Medium

Description:

In `PbftViewManager::ReceivedBroadcastViewChangeMessage`, any committee member can send a `BroadcastViewChangeMessage` with an arbitrarily large `old_view_number`. Each distinct `old_view_number ≥ current_view_number` causes

```
TransientViewData& view_data = GetTransientView(message.old_view_number);
```

to allocate a new entry in the unbounded `transient_views` map. A malicious replica can therefore spam ever-increasing view numbers (`current_view_number+1`, `+2`, `+3`, ...) and force every honest node to consume unbounded memory and CPU processing those stale entries on every tick or view-change pass.

Affected Code

```
void PbftViewManager::ReceivedBroadcastViewChangeMessage(BroadcastViewChangeMessage message) {
    // ...
    // No bound on message.old_view_number here:
    TransientViewData& view_data = GetTransientView(message.old_view_number);
    view_data.received_broadcast_view_change_messages[*sender] = message;
}

TransientViewData& PbftViewManager::GetTransientView(ViewNumber view_number) {
    if (!transient_views.count(view_number)) {
        // Allocates unboundedly for every new view_number
        transient_views.emplace(view_number, ...);
    }
    return *transient_views[view_number];
}
```

Impact

- Memory exhaustion as `transient_views` grows without bound.
- CPU exhaustion on each `Tick()` and view-change scan over all entries.
- Consensus stall or node crash, leading to Denial-of-Service of the entire PBFT engine.

Assets:

- `somnia/pbft/pbft_view_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Severity: Medium

Recommendations

Remediation:

- Bound “future” view numbers: refuse to allocate or record any `old_view_number` more than `current_view_number + MAX_FUTURE_VIEWS` (e.g. `f+1` or a small constant).
- Evict stale entries: periodically garbage-collect or cap the size of `transient_views`.
- Rate-limit per-sender allocations: only one new entry per sender per window.

Resolution:

The finding is fixed in the commit [fb5d24a](#).

A bounds check was added to the `PbftViewManager::ReceivedBroadcastViewChangeMessage` function.

[F-2025-10425](#) - PBFT Parameters Lack Bounds Validation Allowing DoS via Misconfiguration - Low

Description:

The `pbft_parameters.h` file defines a large collection of timeout and interval parameters (e.g., `checkpoint_broadcast_interval`, `new_view_max_timeout`, `target_request_batch_interval`, etc.) without any runtime or compile-time bounds checking. An adversary or misconfigured node could supply extreme values, either very small (triggering constant view-changes and flooding) or excessively large (stalling consensus indefinitely). Because these parameters drive critical timers in the PBFT protocol, unvalidated values can be abused to mount DOS attacks against the network or stall the progress of consensus.

Here's what the issue looks like:

```
struct PbftLocalParameters {
    // <- no clamp here! user-supplied checkpoint_broadcast_interval
    RealClock::duration checkpoint_broadcast_interval = std::chrono::seconds(10);

    // <- no clamp here
    RealClock::duration sent_request_initial_broadcast_timeout = std::chrono::seconds(4);

    // <- no clamp here
    RealClock::duration sent_request_max_broadcast_timeout = std::chrono::seconds(32);

    // ...

    // <- no clamp here
    RealClock::duration view_change_initial_rebroadcast_interval = std::chrono::seconds(4);

    // <- no clamp here
    RealClock::duration view_change_max_rebroadcast_interval = std::chrono::seconds(32);

    // <- no clamp here
    RealClock::duration max_timestamp_tolerantion = std::chrono::seconds(15);

    // ...and on and on for every field in this struct.
```

Assets:

- `somnia/pbft/pbft_parameters.h` [<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate:

3/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: It's advised to introduce explicit minimum and maximum bounds for every PBFT timing parameter, both at compile time and at runtime initialization.

For example:

```
// in pbft_parameters.h constructor or factory
checkpoint_broadcast_interval = clamp(
    user_config.checkpoint_broadcast_interval,
    std::chrono::milliseconds(X),
    std::chrono::seconds(X));
```

Repeat for all intervals and timeouts, choosing sane lower/upper limits based on network round-trip characteristics and expected block cadence.

Resolution: The client has **accepted** the issue and stated:

We want to keep these flexible in case of an emergency advice / changes we may need to ask validators to rollout with. These flags are not documented or intended to be changed by anyone so we would consider a node which finds and changes these themselves to be Byzantine. I don't think we should add the complexity of those bounds seeing as these are internal flags not expected to be changed by a validator.

Evidences

[F-2025-10758](#) - Lack of Rate-Limiting on Signature Verification - Low

Description:

The PBFT implementation processes incoming messages and typically verifies signatures early in the handling path (e.g., `ValidateAndGetSender()` methods in message structs like `PrePrepareMessage`, `BroadcastViewChangeMessage`, `LeaderViewChangeMessage`).

There is no explicit per-peer or global rate-limiting mechanism observed at the message ingress points or before cryptographic signature verifications are performed within the PBFT message handling logic.

Impact

Without rate-limiting, a malicious actor (or a misbehaving peer) could send a high volume of messages, each requiring signature verification. Even if the signatures are invalid and the messages are eventually discarded, the cumulative CPU cost of performing these verifications can be substantial. This can lead to a denial-of-service (DoS) condition by overwhelming the CPU resources of a target node, thereby preventing it from processing legitimate consensus messages, participating in rounds, or serving other critical functions.

Note: We acknowledge that rate-limiting may exist at the networking layer, which is outside this audit's scope. However, application-level protection is still crucial. Network-level defenses typically lack the context to penalize a specific validator `Address` for sending bad messages. A defense-in-depth approach, with protection at both the network and PBFT layers, is recommended for robust DoS resilience.

1. `PrePrepareMessage` Handling:

- Signature verification is called here:

```
// File: pbft_round_manager.cc
// Method: PbftRoundManager::ReceivedPrePrepareMessage
Optional<Address> sender = message.ValidateAndGetSender(); // Signature verification
if (!sender) {
    // Invalid signature
    return;
}
```

- **Missing Check:** Before the line `Optional<Address> sender = message.ValidateAndGetSender();`, there is no code to check if the source of the message (if identifiable before signature validation, e.g., by network ID, or if a persistent sender reputation is

maintained) has sent too many messages or too many invalid messages recently.

2. `BroadcastViewChangeMessage` Handling:

- Signature verification is called here:

```
// File: pbft_view_manager.cc
// Method: PbftViewManager::ReceivedBroadcastViewChangeMessage
Optional<Address> sender = message.ValidateAndGetSender(); // Signature verification
if (!sender) {
// Invalid signature
return;
}
```

- **Missing Check:** Similar to the `PrePrepareMessage`, there's no rate-limiting logic applied to the sender or source before `message.ValidateAndGetSender()` is invoked.

3. `LeaderViewChangeMessage` Handling:

- Signature verification is called here:

```
// File: pbft_view_manager.cc
// Method: PbftViewManager::ReceivedLeaderViewChangeMessage
Optional<Address> sender = message.ValidateAndGetSender(); // Signature verification
if (!sender) {
// Invalid signature
return;
}
```

- **Missing Check:** No rate-limiting is performed before `message.ValidateAndGetSender()`.

4. `NewViewMessage` Handling:

- Signature verification is called here:

```
// File: pbft_view_manager.cc
// Method: PbftViewManager::ReceivedNewViewMessage
Optional<Address> sender = message.ValidateAndGetSender(); // Signature verification
if (!sender) {
// Invalid signature
return;
}
```

- **Missing Check:** No rate-limiting before `message.ValidateAndGetSender()`.

Assets:

- `somnia/pbft/pbft_round.cc` [<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]
- `somnia/pbft/pbft_round_manager.cc` [<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

repo/tree/main]

- somnia/pbft/pbft_byzantine_test/pbft_view_manager.cc

[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

- somnia/pbft/pbft_byzantine_test/pbft_checkpoint_manager.cc

[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Severity: Low

Recommendations

Remediation:

Implement a rate-limiting mechanism for incoming messages, particularly from known peers. This could involve:

1. **Per-Peer Message Limits:** Track the rate of incoming messages (and specifically messages failing signature validation) from each known peer address.
2. **Exponential Backoff/Temporary Bans:** If a peer exceeds defined thresholds for message frequency or invalid signature attempts, apply an exponential backoff or temporarily de-prioritize/ignore messages from that peer.
3. **Global Limits:** Consider global limits on the rate of new messages requiring verification that the node will process in a given time window to protect overall CPU capacity.
4. **Network Layer Protection:** Where possible, leverage lower-level network protections (e.g., firewalls, DDoS mitigation services) but also ensure application-level resilience. It is important to design these rate-limiting mechanisms carefully to avoid inadvertently penalizing honest peers during legitimate periods of high network activity or temporary connectivity issues.

Resolution:

The finding is **Accepted** by the client.

The client has confirmed they have implemented per-IP rate limiting at the network layer, which was out of scope for this audit. This control effectively mitigates the risk of a DoS attack via signature verification spam.

[F-2025-10784](#) - Clock Skew Can Induce Unnecessary Potential False View-Changes - Low

Description:

The Somnia PBFT implementation validates the timestamps of `RequestBatch` objects.

1. When a node considers sending a `PrepareMessage`, `PbftRoundManager::CanSendPrepareMessage` checks if the timestamp of the received `PrePrepareMessage`'s `RequestBatch` is within `parameters.local_parameters.max_timestamp_toleration` (default 15 seconds) of the node's current local time. This check is skipped if `round.skip_timestamp_validation` is true (e.g., during view changes for already prepared rounds).
2. When a leader proposes a new batch, its timestamp is also implicitly constrained by `PbftRoundManager::ReceivedPrePrepareMessage`, which checks if the batch timestamp is not too early relative to its sequence number and the network start time ($(\text{network_setup.network_start_time} + \text{message.request_batch.sequence_number} * \text{parameters.local_parameters.target_request_batch_interval})$). If nodes' local clocks drift significantly beyond the `max_timestamp_tolerance`, honest replicas might reject valid proposals from an honest (but differently clocked) leader.

Impact

Rejection of valid proposals due to timestamp mismatches (caused by clock skew exceeding tolerance) can lead to rounds failing to commit. This, in turn, can trigger timeout mechanisms within `PbftViewManager` (e.g., "timeout between successive request batch commitments" or "timeout while waiting for any request to execute") that result in unnecessary view changes. Frequent view changes can degrade overall network performance, throughput, and liveness.

Exploitation Scenario

This is primarily an operational risk rather than a directly exploitable vulnerability by an external attacker (unless the attacker can manipulate a node's clock). If a significant portion of the validator set, or a sequence of leaders, experience clock drift, the network could become unstable due to frequent view changes initiated by honest but out-of-sync nodes.

Snippet 1: The Primary Timestamp Tolerant Check

This snippet from `pbft_round_manager.cc` shows the direct comparison between the node's local clock and the leader's proposal timestamp.

1. `PbftRoundManager::CanSendPrepareMessage` ~Line: 199 - 208

```
bool PbftRoundManager::CanSendPrepareMessage(const PbftRound& round
) const {
// ... (other checks) ...

if (!round.skip_timestamp_validation) {
// This check is performed unless explicitly skipped (e.g., during
a view change).

// The following line calculates the absolute time difference between:
// 1. 'current_time': The validating node's own local clock time.
// 2. 'round.request_batch->timestamp': The timestamp the leader put
on the proposal.
auto time_difference_to_local =
std::chrono::abs(current_time - UnixMillisToTimePoint(round.request
_batch->timestamp)); <<< This line shows use of the node's local ti
me for this check

// This 'if' statement then compares the calculated difference again
st the configured
// tolerance ('max_timestamp_tolerantion').
if (parameters.protocol_parameters.enforce_request_batch_timestamps
&&
time_difference_to_local > parameters.local_parameters.max_timestam
p_tolerantion) {

// If the difference is greater than the tolerance (e.g., > 15 seco
nds), this
// comment is reached, and the function returns 'false'. This cause
s the node
// to reject the leader's proposal, which can lead to a view change
if it persists.

return false;
}
```

Snippet 2: The `max_timestamp_tolerantion` Parameter Definition

This snippet from `pbft_parameters.h` shows where the default 15-second tolerance is defined.

`PbftLocalParameters::max_timestamp_tolerantion` ~Line: 106

```
struct PbftLocalParameters {
// ...

// This is the configured tolerance used in the check from Snippet
1. It defaults
// to 15 seconds. If the time difference between nodes exceeds this
, proposals are rejected.
HELP(
"This is the maximum allowed absolute difference between a proposed
request batch's "
"timestamp and the nodes local clock. If this is exceeded, the node
will not vote on this "
"request batch, and the view will change to a new leader.")
RealClock::duration max_timestamp_tolerantion = std::chrono::seconds
(15);

// ...
};
```

Snippet 3: The Proposal Schedule Adherence Check

This snippet from `pbft_round_manager.cc` shows the second timestamp check, which ensures the leader is not proposing batches too early for their sequence number.

`PbftRoundManager::ReceivedPrePrepareMessage` ~Line: 415 - 422

```
// File: pbft_round_manager.cc
void PbftRoundManager::ReceivedPrePrepareMessage(PrePrepareMessage
message) {
// ...

RealClock::time_point request_batch_time = UnixMillisToTimePoint(me
ssage.request_batch.timestamp);

// This check ensures the leader is not proposing batches "too fast
" or out of schedule.
// It calculates the minimum expected time for a given sequence num
ber based on the
// network start time and a fixed interval per block.
if (request_batch_time < network_setup.network_start_time +
message.request_batch.sequence_number *
parameters.local_parameters.target_request_batch_interval) {

// If the proposal's timestamp is earlier than the calculated minim
um, it is rejected.
// This is another way clock skew (specifically a leader's clock be
ing far behind)
// or malicious behavior could cause proposal rejection.
// This request batch has a timestamp too soon, given it's sequence
number. The leader is
// proposing request batches too quickly.
return;
}
```

Assets:

- `somnia/pbft/pbft_parameters.h` [<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]
- `somnia/pbft/pbft_round_manager.cc` [<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 2/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation:

1. **Operational Mandate for Clock Synchronization:** Clearly document and enforce that all validator nodes participating in the Somnia network must maintain their system clocks

synchronized to a reliable time source (e.g., using NTP - Network Time Protocol).

2. **Monitoring:** Implement monitoring for validator nodes to detect and alert on significant clock drift relative to a global time standard or amongst committee members.
3. **Review** `max_timestamp_tolerant` : The 15-second default is a common value. However, it should be reviewed in the context of the target operational environment for Somnia nodes. If nodes are expected to operate in environments with potentially higher clock drift, this tolerance might need to be carefully increased, balancing the risk of false view changes against the risk of accepting batches with overly stale timestamps.

Resolution:

The finding is Accepted by the client.

The client's minimum hardware specifications require server-grade infrastructure with proper NTP time synchronization, which mitigates the risk of significant clock skew. The finding will be kept to document the protocol's dependency on accurate clock synchronization.

[F-2025-10837](#) - Configuration Risk Allowing Permanent Liveness Failure - Low

Description: The `PbftLocalParameters` struct contains a boolean parameter named `should_continue_on_divergence`. This parameter is intended for testing and debugging, specifically for use in the project's fuzzer. Its purpose is to prevent the node from crashing when it detects a `sequence_commitment` mismatch.

However, this parameter is not guarded by compile-time flags (e.g., `#ifndef NDEBUG`) and can be enabled in a production build. If this flag is misconfigured to `true` on a production node, it disables the critical safety assertion that would normally terminate the process upon detecting a state divergence.

Affected Code

File: `pbft_committed_request_batch_manager.cc` @ Line 290-311

The vulnerability is located in the `TrySendRequestBatchesToInterface` function. This code block is responsible for verifying that a request batch's `sequence_commitment` matches the node's locally expected commitment before it is passed to the application layer.

```
void PbftCommittedRequestBatchManager::TrySendRequestBatchesToInterface() {
    // ...
    if (request_batch.sequence_commitment != *sequence_commitment) {
        spdlog::critical(
            "Replica {} had a different local sequence commitment... This is a fatal unexpected state divergence.",
            // ... (logging details)
        );

        // PROBLEM: This block bypasses the critical safety check if true.
        if (parameters.local_parameters.should_continue_on_divergence) {
            return; // The node silently ignores the mismatch and continues.
        }

        // ...

        // This is the essential safety mechanism that is bypassed.
        // In a default configuration, it terminates the node to prevent corruption.
        RELEASE_ASSERT(false);
    }
    //...
}
```

Impact

While enabling `should_continue_on_divergence` in production does not cause the node to commit an invalid block (a safety violation), it introduces a liveness failure.

- **Permanent Consensus Stall:** When the `sequence_commitment` mismatch is detected, the function `return`s early without advancing the sequence number. This traps the node in an infinite loop on every subsequent `Tick()`, where it attempts to process the same invalid block forever. The node is effectively "bricked" and will never process another transaction until it is manually restarted with a corrected configuration.
- **Degradation of Network Fault Tolerance:** A permanently stalled node is, from the network's perspective, offline. A single misconfiguration can therefore reduce the network's resilience from `f` faulty nodes to `f-1`, increasing the risk that other, unrelated failures could halt the entire blockchain.

Risk Analysis

While the default value of this parameter is `false`, its existence as a runtime-configurable parameter in production code constitutes a significant operational risk. Dangerous debugging and testing flags should not be present or configurable in production builds. The client has acknowledged this behavior as a liveness failure that falls into the category of operator misconfiguration risk.

Assets:

- `somnia/pbft/pbft_committed_request_batch_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation:

The primary recommendation is to ensure this dangerous test-only feature cannot be enabled in a production environment.

1. Use Compile-Time Guards: The safest remediation is to wrap the entire bypass check in a preprocessor directive that removes it from release builds. This ensures the bypass can only ever exist in debug or test builds.

```
// --- RECOMMENDED CHANGE ---  
// in pbft_committed_request_batch_manager.cc  
  
#ifndef NDEBUG  
// This entire block will only be compiled in debug/test builds.  
if (parameters.local_parameters.should_continue_on_divergence) {  
    return;  
}  
#endif  
  
RELEASE_ASSERT(false);
```

2. Explicit Naming Convention: If the parameter must remain, rename it to more clearly indicate its danger, for example:

```
DANGEROUS_UNSAFE_TESTING_ignore_commitment_divergence.
```

3. Documentation Warnings: Add prominent warnings in all relevant configuration guides and documentation, explicitly stating that this flag must never be enabled in a production environment as it will break consensus safety.

Resolution:

The client has acknowledged this behavior but accepts the operational risk, noting that they have external monitoring (i.e., block height alerts) that would detect this specific failure mode.

F-2025-10873 - Excessive CPU in Tick() Loop - Low

Description:

The `Tick()` method, on every invocation, scans from the first non-checkpointed sequence up to the first missing state. In a healthy system, this is small, but an attacker who withholds state can force the loop to iterate over every checkpoint-eligible `sequence_number` on each tick, burning CPU.

Affected Code:

```
for (SequenceNumber sequence_number = GetFirstNonCheckpointedSequenceNumber(); ; sequence_number++) {
    if (!ShouldCheckpointSequenceNumber(sequence_number)) {
        continue;
    }
    if (!persistent_state.HasSequenceState(sequence_number)) {
        break;
    }
    // ... heavy work ...
}
```

Attack Scenario

- **Step 1:** A malicious proposer can either stop proposing entirely or submit blocks with empty or malformed payloads. This increases the number of invalid checkpoints other nodes must process, effectively polluting the consensus with computationally expensive payloads.
- **Step 2:** `Tick()` is called periodically (e.g., every 100ms).
- **Step 3:** Each `Tick()` invocation scans all uncheckpointed sequences (e.g., 10,000+ iterations).
- **Result:** CPU burns cycles on:
 - Hash computations (`TryFillCumulativeRequestBatchHash`).
 - Signature validations (`TryBuildCheckpointQuorumCertificate`).
 - Network I/O (`TrySendCheckpointMessage`).

Impact:

- **CPU exhaustion DoS:** starves other consensus work.

Assets:

- `somnia/pbft/pbft_byzantine_test/pbft_checkpoint_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Severity: Low

Recommendations

Remediation: Limit the number of iterations per tick (e.g. process at most K sequence numbers before returning), or track an upper bound and resume in the next tick.

You can either add:

Loop Bound (Simplest Fix)

```
constexpr int kMaxSequencesPerTick = 100; // Configurable
for (SequenceNumber seq = GetFirstNonCheckpointedSequenceNumber(),
     i = 0;
     i < kMaxSequencesPerTick; seq++, i++) {
    if (!ShouldCheckpointSequenceNumber(seq)) continue;
    if (!persistent_state.HasSequenceState(seq)) break;
    // ...
}
```

Exponential Backoff

```
auto start_time = std::chrono::steady_clock::now();
for (SequenceNumber seq = GetFirstNonCheckpointedSequenceNumber();
     seq++;) {
    if (std::chrono::steady_clock::now() - start_time > 10ms) break; //
    Time-bound
    // ...
}
```

Prioritization

Skip non-urgent work if behind:

```
if (GetFirstNonCheckpointedSequenceNumber() > kCatchupThreshold) {
    TryBroadcastLatestCheckpointCertificate(); // Focus on sync first
    return;
}
```

Resolution: The finding is **Accepted** by the client.

The client has confirmed that in their production environment, the PBFT network is bounded to 3000 sequence numbers per epoch. This provides a practical upper limit for the `Tick()` loop and mitigates the risk of a CPU DoS. The client has agreed to add an additional check for robustness in non-production environments. Given that the risk is mitigated in production by existing architectural constraints, the finding is accepted.

Evidences

Proof of Concept (PoC):

Reproduce:

Create a standalone test file (`pbft_tick_poc.cpp`) that replicates the `Tick()` logic in a minimal environment:

```
// pbft_tick_poc.cpp
#include <chrono>
#include <iostream>
#include <unordered_map>

// we simulated minimal parts of the original code
struct SequenceNumber { uint64_t n; };
struct PbftPersistentState {
    std::unordered_map<uint64_t, bool> sequence_states;
    bool HasSequenceState(SequenceNumber seq) const {
        return sequence_states.count(seq.n) > 0;
    }
    void InsertGap(SequenceNumber start, uint64_t count) {
        for (uint64_t i = 0; i < count; ++i) sequence_states[start.n + i] = true;
    }
};

class PbftCheckpointManager {
    PbftPersistentState persistent_state;
    SequenceNumber last_checkpointed_sequence_number{0};
public:
    void Tick() {
        auto start_time = std::chrono::steady_clock::now();
        for (SequenceNumber seq{last_checkpointed_sequence_number.n + 1};;
            seq.n++) {
            if (!persistent_state.HasSequenceState(seq)) break;
            // Simulate heavy work (e.g., crypto, network)
            volatile int x = 0; // Force CPU work
            for (int i = 0; i < 1000; ++i) x++;
        }
        auto duration = std::chrono::duration_cast<std::chrono::millisecond>(
            std::chrono::steady_clock::now() - start_time
        );
        std::cout << "Tick() took " << duration.count() << "ms\n";
    }
    void InsertGap(uint64_t count) {
        persistent_state.InsertGap(SequenceNumber{last_checkpointed_sequence_number.n + 1}, count);
    }
};

int main() {
    PbftCheckpointManager manager;

    // Simulate attack: create 10,000 uncheckpointed sequences
    manager.InsertGap(10'000); // You can also try higher gaps to see time varies

    // Measure Tick() CPU usage
    manager.Tick();
    return 0;
}
```

Compile and Run

```
return round_config->Create(*node_context);
```

Each manager's constructor takes a `PbftNodeContext&` and (almost certainly) stores it for later use. If the factory is destroyed (or

moved-from) while any of its created managers remain alive, those managers will hold a dangling reference to freed memory, leading to use-after-free and undefined behavior.

```
// pbft_node_manager_factory.h
struct PbftManagerFactory {
// ...
Optional<PbftNodeContext> node_context; // Stores Temporary Context
Owned by Factory

void InitialiseForNode(PbftNodeContext in_node_context) {
RELEASE_ASSERT(!node_context.has_value());
node_context.emplace(std::move(in_node_context));
}

std::unique_ptr<PbftRoundManager> CreateRoundManager() {
RELEASE_ASSERT(node_context.has_value());
// ← Passing a reference into the factory's internal Optional
return round_config->Create(*node_context); // Manager gets a refer
ence
}
// ... same pattern for CreateViewManager, CreateCheckpointManager, C
reateRequestSender
};
```

Impact

- **Use-After-Free / Crash:** Any manager method that dereferences its stored `PbftNodeContext&` after the factory is destroyed will crash the process.
- **Memory Corruption:** If the freed memory is reallocated, managers may read or write garbage, silently corrupting in-memory PBFT state.
- **Protocol Violation:** Corrupted context data can violate PBFT safety and liveness invariants, potentially halting or forking the network.

Assets:

- somnia/pbft/pbft_node_manager_factory.h
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 1/5

Severity: Low

Recommendations

Remediation: Ensure that the context outlives all managers, either by:

Shared Ownership: Store `node_context` as

`std::shared_ptr<PbftNodeContext>` in the factory, and pass that `shared_ptr` into every manager so they share ownership.

```
// In PbftManagerFactory:
std::shared_ptr<PbftNodeContext> node_context; // Shared ownership

void InitialiseForNode(PbftNodeContext ctx) {
    node_context = std::make_shared<PbftNodeContext>(std::move(ctx));
}

std::unique_ptr<PbftRoundManager> CreateRoundManager() {
    return round_config->Create(node_context); // Pass shared_ptr
}
```

Managers would then store:

```
std::shared_ptr<PbftNodeContext> context_; // Safe lifetime
```

Value Semantics: Have each manager make its own copy of `PbftNodeContext` rather than holding a reference.

```
// In manager constructors:
PbftNodeContext context_; // Store by value

// Factory passes a copy:
return round_config->Create(*node_context); // Implicit copy
```

Explicit Contract (Stop-gap): Document that `PbftManagerFactory` must outlive all created managers and enforce this via code reviews or lints.

Resolution:

The finding is **Accepted** by the client.

The client has confirmed that the `PbftManagerFactory`'s lifetime is tied to the `PbftNode` object that owns it. Because the context it holds refers to members of that same node, the references will always be valid, and this is not a risk in the current implementation.

Evidences

Proof of Concept (PoC):

Reproduce:

Create a file `poc_dangling_reference.cpp` with with the code below:

```
// poc_dangling_reference.cpp
#include <memory>
#include <optional> // ← Add this
#include <iostream>

// Simulated context
struct PbftNodeContext {
    int critical_data = 42;
};

// Manager that holds a reference to the context
```

```

class PbftRoundManager {
public:
    explicit PbftRoundManager(PbftNodeContext& ctx) : context_(ctx) {}
    void Tick() {
        // UB if ctx has been destroyed:
        std::cout << "Accessing context data: " << context_.critical_data <
        < "\n";
    }
private:
    PbftNodeContext& context_;
};

// Generic factory config stub
struct BasePbftManagerConfiguration {
    virtual std::unique_ptr<PbftRoundManager>
    Create(PbftNodeContext& ctx) {
        return std::make_unique<PbftRoundManager>(ctx);
    }
    virtual ~BasePbftManagerConfiguration() = default;
};

// The buggy factory
struct PbftManagerFactory {
    void InitialiseForNode(PbftNodeContext ctx) {
        node_context_.emplace(std::move(ctx));
    }

    std::unique_ptr<PbftRoundManager> CreateRoundManager() {
        return std::make_unique<BasePbftManagerConfiguration>()
        ->Create(*node_context_);
    }

private:
    std::optional<PbftNodeContext> node_context_;
};

int main() {
    auto factory = std::make_unique<PbftManagerFactory>();

    { // 1) Initialize the factory with a context
        PbftNodeContext ctx;
        factory->InitialiseForNode(std::move(ctx));
    } // Original `ctx` is gone, but optional has its own copy

    auto manager = factory->CreateRoundManager(); // manager holds refe
    rence to optional's copy

    factory.reset(); // 2) Destroys optional+context → manager.context_
    now dangling

    std::cout << "About to call Tick() on a manager with a dangling con
    text:\n";
    manager->Tick(); // UB: likely crash or sanitize

```

[See more](#)

[F-2025-10999](#) - Uninitialized collaborator pointers in PbftViewManager - Low

Description:

`PbftViewManager` holds four raw pointers, `round_manager`, `checkpoint_manager`, `received_request_manager`, and `timeout_manager`, which are only set inside the `Initialise(...)` method. Nearly every other public API (e.g. `Tick()`, `SequenceNumberCommitted()`, view-change handlers, `TimeoutCurrentView()`, etc.) dereferences one or more of these pointers without any release-build guard on `is_initialised`. In production builds (`-DNDEBUG`), the only check (`DEBUG_ASSERT(is_initialised)`) is compiled out, so any invocation before `Initialise()` leads to a null-pointer dereference and undefined behavior.

Affected Code

```
// pbft_view_manager.h
struct PbftViewManager {
// ...
struct PbftRoundManager* round_manager = nullptr;
struct PbftCheckpointManager* checkpoint_manager = nullptr;
struct PbftReceivedRequestBatchManager* received_request_manager =
nullptr;
struct PbftTimeoutManager* timeout_manager = nullptr;
bool is_initialised = false;
// ...
};

// pbft_view_manager.cc
void PbftViewManager::Tick(RealClock::time_point now) {
    DEBUG_ASSERT(is_initialised); // vanishes in release
    current_time = now;

    // → Derefs checkpoint_manager and round_manager without any release-time check
    for (ViewNumber v = current_view_number; v++; v++) {
        if (persistent_state.HasViewState(v)) { /*...*/ }
        else break;
    }
    TrySendLeaderViewChangeMessage(); // may deref timeout_manager
    TryBroadcastViewChangeMessage(); // may deref timeout_manager
    // ... and dozens of other methods all dereference these pointers unguarded
}

void PbftViewManager::SequenceNumberCommitted(SequenceNumber sequence_number) {
    // → Derefs timeout_manager without init check
    timeout_manager->OnNewSequenceNumberCommitted();
    next_sequence_number_expiry_time =
        current_time + timeout_manager->CurrentNewSequenceNumberTimeout();
}
```

Impact

- **Denial-of-Service:** Any stray call to `Tick()`, `SequenceNumberCommitted()`, or any view-change handler before `Initialise()` will crash the process in production.
- **Undefined Behavior:** If the uninitialized pointer happens to be non-null garbage, memory corruption or other unpredictable

faults may ensue.

- **Consensus Stall:** A crashed or corrupted view manager halts PBFT progress, blocking block finalization and threatening network liveness.

While the current codebase is unlikely to suffer from this issue, since the `PbftNode` constructor creates the `PbftViewManager` and immediately calls `Initialise()`, this could change in future iterations of the code. Therefore, the issue is raised as a preventive measure.

Assets:

- `somnia/pbft/pbft_byzantine_test/pbft_view_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]
- `somnia/pbft/pbft_view_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Impact Rate: 4/5

Severity: Low

Recommendations

Remediation: At the top of every public method, enforce initialization in release builds:

```
// In every public API (Tick, SequenceNumberCommitted, view handler  
S...):  
RELEASE_ASSERT(is_initialised && round_manager && checkpoint_manage  
r &&  
received_request_manager && timeout_manager);
```

Replace raw pointers with `gsl::not_null<...*>` or `std::optional<std::reference_wrapper<...>>`. This converts misuse into a compile-time error rather than runtime UB.

Resolution: The finding is **Accepted** by the client.

It has been acknowledged, but will not be implementing initialization guards. They state that the `PbftNode` is the sole entry point for creating this manager and its architecture guarantees correct initialization, making explicit guards unnecessary code pollution in their specific use case.

F-2025-10528 - Missing Committee Membership Validation - Info

Description:

In Somnia's PBFT implementation, `PrepareMessage` and `CommitMessage` handlers validate that the sender is a member of the active committee using `epoch_data.IsInCommittee(...)`. However, the `ReceivedPrePrepareMessage()` function does not enforce this check. It only validates that the sender matches the expected leader address:

pbft_round.manager.cc ≈ lines 399-408

```
Optional<Address> sender = message.ValidateAndGetSender();  
if (!sender) return;  
  
if (*sender != epoch_data.ConsensusLeaderForView(message.view_number)) return;
```

Impact:

A node that is no longer in the current committee, but still possesses a previously valid leader key, could craft or replay a valid `PrePrepareMessage` and have it processed by honest replicas. This bypasses the expected committee membership guard and creates a risk of:

- Injecting unauthorized proposals
- Causing round state divergence
- Triggering unnecessary view changes

Exploitation Potential:

This gap could be exploited during committee rotation or if replicas are operating on outdated epoch data. An adversary may replay a message signed by an old leader whose key is still valid but no longer part of the current committee.

Assets:

- somnia/pbft/pbft_round_manager.cc
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Accepted

Classification

Severity:

Info

Recommendations

Remediation:

Add a committee membership check directly after verifying the sender identity and leader role:

```
if (!epoch_data.IsInCommittee(*sender)) {  
    // This sender is not part of the current committee.  
    return;  
}
```

Final placement should be after:

```
if (*sender != epoch_data.ConsensusLeaderForView(message.view_number)) return;
```

This brings `ReceivedPrePrepareMessage()` in line with `ReceivedPrepareMessage()` and `ReceivedCommitMessage()`, ensuring only current committee members are permitted to participate in consensus.

Resolution:

The finding is **Accepted** by the client.

The client has clarified that in their implementation, the consensus committee is immutable for an entire epoch. Each epoch runs as a separate and distinct `PbftNode` instance. This architectural design makes the "committee drift" scenario impossible, rendering the suggested check redundant. The finding will be kept as Informational to document this key design invariant.

[F-2025-10897](#) - Implicit Copyability of PbftCheckpointManager - Info

Description: Both `PbftCommittedRequestBatchManager` and `PbftCheckpointManager` own non-owning references and raw pointers (e.g. to `PbftInterface`, `PbftMessageSender`, `PersistentState`, `PbftViewManager`, etc.) yet do not declare or delete copy/move constructors or assignment operators. The compiler will synthesize defaults that simply bit-copy every member, causing two instances to alias the same underlying resources and state.

Affected Code

```
// PbftCommittedRequestBatchManager (somnia/pbft/pbft_committed_request_batch_manager.h)
struct PbftCommittedRequestBatchManager : ResourceSystemBase::ResourceSystemMessageSender {
    //...
    PbftInterface& interface;
    PbftMessageSender& message_sender;
    PbftReceivedRequestManager& received_request_manager;
    struct PbftRequestSender* request_sender;
    struct PbftTimeoutManager* timeout_manager;
    //...
};

// PbftCheckpointManager (somnia/pbft/pbft_checkpoint_manager.h)
struct PbftCheckpointManager {
    // ... many pointer & reference members ...
    PbftCheckpointManager(PbftNodeContext& context);
    virtual ~PbftCheckpointManager() = default;
    // no explicit delete of copy/move constructors
};
```

Impact

- **Safety:** Two instances can concurrently invoke the same `interface` methods, breaking the in-order execution guarantee.
- **Availability:** If one instance resets or tears down shared resources, the other may dereference dangling pointers, causing crashes (DoS).

Assets:

- `somnia/pbft/pbft_committed_request_batch_manager.cc`
[<https://github.com/metaversal-foundation/somnia-audit-repo/tree/main>]

Status:

Fixed

Classification

Severity:

Info

Recommendations

Remediation:

- For each manager class, explicitly delete copy and move operations to enforce single ownership of those reference/pointer members.
- The virtual destructor triggers the Rule of Five, making explicit deletions mandatory (as per C++ Core Guidelines [C.21](#)).

This sample fix below for `PbftCheckpointManager`:

```
PbftCheckpointManager(const PbftCheckpointManager&) = delete;  
PbftCheckpointManager& operator=(const PbftCheckpointManager&) = delete;  
PbftCheckpointManager(PbftCheckpointManager&&) = delete;  
PbftCheckpointManager& operator=(PbftCheckpointManager&&) = delete;
```

Addresses the Issue by:

1. **Preventing Copies:** Ensures only one instance manages critical consensus state.
2. **Explicit Intent:** Clearly communicates non-copyability in the API.
3. **Future-Proofing:** Guards against refactoring errors.

Resolution:

The finding is fixed in the commit [fb5d24a](#).

The `NON_COPYABLE_OR_MOVABLE` macro was added.

[F-2025-10976](#) - Unbounded Growth of pending_sent_requests leading to CPU & Memory DoS - Info

Description:

`PbftRequestSender::SendRequest()` adds every new payload to

`pending_sent_requests` :

```
//somnia/pbft/pbft_request_sender.cc
pending_sent_requests[request.CalculatePayloadHash()] = pending_sen
t_request;
```

There is no cap on the number of distinct hashes stored. On each `Tick()`, the code iterates over all entries and invokes `TryBroadcastRequestMessage()`, performing cryptographic and I/O work per entry. An attacker flooding the node with unique payloads can exhaust CPU and memory, stalling consensus.

Affected Code

```
void PbftRequestSender::SendRequest(const RequestPayload& request)
{
    // ...
    pending_sent_requests[request.CalculatePayloadHash()] = pending_sen
t_request;
}

void PbftRequestSender::Tick(RealClock::time_point now) {
    // ...
    for (auto& [payload_hash, pending_request] : pending_sent_requests)
    {
        // cryptographic checks, then broadcast
        TryBroadcastRequestMessage(pending_request);
    }
}
```

Impact

- **CPU Exhaustion:** $O(N)$ broadcast work every tick (e.g. 10 000+ entries $\times$ timer frequency).
- **Memory Exhaustion:** Unbounded map growth (OOM).
- **Consensus Stall:** Node unable to process real requests or advance blocks.

Conclusion

The finding correctly notes that the `pending_sent_requests` map can grow without a defined bound. However, its trigger, the `SendRequest()` method, is called by internal components that were outside this audit's scope.

Therefore, while the lack of an internal size limit poses a resilience risk against potential bugs or cascading system failures, we did not identify a direct, exploitable attack path within the consensus module itself.

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: Enforce a configurable cap on `pending_sent_requests` size. Reject or evict old entries when the cap is reached:

```
constexpr size_t kMaxPendingRequests = 10'000; // Configurable

void SendRequest(const RequestPayload& request) {
    if (pending_sent_requests.size() >= kMaxPendingRequests) {
        // Evict oldest entry (or random)
        pending_sent_requests.erase(pending_sent_requests.begin());
    }
    // Add new entry
    pending_sent_requests.emplace(request.CalculatePayloadHash(), pending_sent_request);
}
```

Limit requests per peer:

```
if (requests_from_peer[peer]++ > kMaxRequestsPerPeer) {
    blacklist(peer);
}
```

Reject requests that cannot be executed immediately:

```
if (interface.GetRequestExecutionStatus(request) == PbftExecutionStatus::CAN_NEVER_BE_EXECUTED) {
    return; // Skip upfront
}
```

Resolution: The finding is **Accepted** by the client.

The `SendRequest()` method is only called by the local node itself and is not exposed to other peers. Therefore, the client does not want to add the complexity of extra rate-limiting for what is considered a trusted, internal-only code path.

Observation Details

Disclaimers

Hacken Disclaimer

The blockchain protocol given for audit has been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in the protocol source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other protocol statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the blockchain protocol.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Blockchain protocols are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the protocol can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited blockchain protocol.

Appendix 1. Severity Definitions

Severity	Description
Critical	Vulnerabilities that can lead to a complete breakdown of the blockchain network's security, privacy, integrity, or availability fall under this category. They can disrupt the consensus mechanism, enabling a malicious entity to take control of the majority of nodes or facilitate 51% attacks. In addition, issues that could lead to widespread crashing of nodes, leading to a complete breakdown or significant halt of the network, are also considered critical along with issues that can lead to a massive theft of assets. Immediate attention and mitigation are required.
High	High severity vulnerabilities are those that do not immediately risk the complete security or integrity of the network but can cause substantial harm. These are issues that could cause the crashing of several nodes, leading to temporary disruption of the network, or could manipulate the consensus mechanism to a certain extent, but not enough to execute a 51% attack. Partial breaches of privacy, unauthorized but limited access to sensitive information, and affecting the reliable execution of smart contracts also fall under this category.
Medium	Medium severity vulnerabilities could negatively affect the blockchain protocol but are usually not capable of causing catastrophic damage. These could include vulnerabilities that allow minor breaches of user privacy, can slow down transaction processing, or can lead to relatively small financial losses. It may be possible to exploit these vulnerabilities under specific circumstances, or they may require a high level of access to exploit effectively.
Low	Low severity vulnerabilities are minor flaws in the blockchain protocol that might not have a direct impact on security but could cause minor inefficiencies in transaction processing or slight delays in block propagation. They might include vulnerabilities that allow attackers to cause nuisance-level disruptions or are only exploitable under extremely rare and specific conditions. These vulnerabilities should be corrected but do not represent an immediate threat to the system.

Appendix 2. Scope

The scope of the project includes the following components from the provided repository:

Scope Details	
Repository	https://github.com/metaversal-foundation/somnia-audit-repo/tree/main
Commit	0af8c04569b626dd5c0cace903998b9ec06ab5a6
Whitepaper	
Requirements	
Technical Requirements	

Components in Scope

The audit focused exclusively on the core C++ implementation of the Somnia consensus protocol contained within the `somnia/pbft/` directory.

In-Scope Files

- **Headers (.h):**

- `pbft_checkpoint_manager.h`
- `pbft_committed_request_batch_manager.h`
- `pbft_interface.h`
- `pbft_node.h`
- `pbft_node_context.h`
- `pbft_node_manager_factory.h`
- `pbft_parameters.h`
- `pbft_persistent_state.h`
- `pbft_received_request_manager.h`
- `pbft_request_sender.h`
- `pbft_round.h`
- `pbft_round_manager.h`
- `pbft_timeout_manager.h`
- `pbft_types.h`
- `pbft_view_manager.h`

- **Sources (.cc):**

- `pbft_checkpoint_manager.cc`
- `pbft_committed_request_batch_manager.cc`
- `pbft_node.cc`
- `pbft_received_request_manager.cc`
- `pbft_request_sender.cc`
- `pbft_round.cc`
- `pbft_round_manager.cc`

- `pbft_timeout_manager.cc`
- `pbft_view_manager.cc`

Core Protocol Review: A thorough review of the C++ implementation of the Practical Byzantine Fault Tolerance (PBFT) protocol.

Consensus Safety Analysis: Focused analysis of the view change mechanism, checkpointing, and round management to identify potential chain forks or liveness failures.

Security Assessment: A review of the code for common C++ vulnerabilities, resource exhaustion vectors, and logic flaws that could compromise the consensus protocol.