Zellic

Aug 15, 2025

Prepared for
James Hunsaker
Category Labs, Inc.

Prepared by
Xuehao Guo
Kuilin Li
Nan Wang
Zellic

Monad

Program Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Category Labs, Inc. from July 7th, 2025 to September 5th, 2025. The assessment encompassed multiple components, including the compiler, consensus, execution, RPC, networking, and database layers. The review was conducted in parallel, with dedicated teams focusing on distinct portions of the codebase. During this engagement, Zellic reviewed Monad's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can the Monad consensus protocol (MonadBFT) ensure safety and liveness in a Byzantine environment?
- Do the consensus-layer message validation mechanisms effectively prevent adversaries from inflating stake via forged votes or timeout certificates?
- Are there resource-exhaustion vectors that could be exploited to mount denial-of-service attacks?
- Do the RLP encoding/decoding implementations correctly handle edge cases and maliciously crafted inputs?
- Can the block sync and networking protocols withstand attacks from malicious peers?
- Are there any discrepancies between consensus-layer and execution-layer transaction validation that, due to the delayed-validation mechanism, could cause computational inconsistencies and lead to network forks?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Monad targets, we discovered 11 findings. Five critical issues were found. Two were of high impact, one was of medium impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Category Labs, Inc. in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	5
🟧 High	2
🟨 Medium	1
🟩 Low	0
⬜ Informational	3

2. Introduction

2.1. About Monad

Category Labs, Inc. contributed the following description of Monad:

> The Monad protocol is an L1 blockchain designed to deliver full EVM compatibility with significant performance improvements. On current (testnet) releases, the client developed by Category Labs has been capable of thousands of tps (transactions per second), 400ms block times and 800ms finality with a globally distributed validator set. Monad's performance derives from optimization in several areas:

> - MonadBFT for performant, tail-fork-resistant BFT consensus
> - RaptorCast for efficient block transmission
> - Asynchronous Execution for pipelining consensus and execution to raise the time budget for execution
> - Parallel Execution for efficient transaction execution
> - MonadDb for efficient state access

> To develop the Monad client software, the engineering team at Category Labs draws upon deep experience from high frequency trading, networking, databases, web3 and academia. For more on Category's ongoing technical work, check out the category.xyz website and follow @category_xyz on Twitter.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the targets.

Architecture risks. This encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system's fundamental security attributes, presumptions, trust mode, and design.

Arithmetic issues. This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

Implementation risks. This encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

Availability. Denial-of-service attacks are another leading issue in custom systems. Issues including but not limited to unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped targets itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Monad Targets

Type	Rust & C++
Platform	Monad

Target	monad-bft
Repository	https://github.com/category-labs/monad-bft ↗
Version	`be342260a8875c6d0ada60857017ec093a04b844`

Programs	`monad-consensus`
	`monad-consensus-types`
	`monad-consensus-state`
	`monad-state`
	`monad-statesync`
	`monad-blocksync`
	`monad-executor`
	`monad-executor-glue`
	`monad-crypto`
	`monad-bls`
	`monad-block-persist`
	`monad-blocktree`
	`monad-chain-config`
	`monad-compress`
	`monad-eth-block-policy`
	`monad-eth-block-validator`
	`monad-eth-ledger`
	`monad-eth-txpool-ipc`
	`monad-eth-txpool-types`
	`monad-eth-types`
	`monad-event-ring`
	`monad-ledger`
	`monad-merkle`
	`monad-multi-sig`
	`monad-node-config`
	`monad-secp`
	`monad-state-backend`
	`monad-twins`
	`monad-types`
	`monad-updaters`
	`monad-validator`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 46.6 person-weeks. This portion of the assessment was conducted by three consultants over the course of 7 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Xuehao Guo
Engineer
guo@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

Nan Wang
Engineer
nan@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 7, 2025	Kick-off call
July 7, 2025	Start of primary review period
August 12, 2025	End of primary review period
September 3, 2025	Commit updated to 2854c7d5
September 8, 2025	End of secondary review period

3. Detailed Findings

3.1. Memory-exhaustion attack via unbounded round voting in consensus

Target	monad-consensus		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Current validation of a `VoteMessage` only checks that `vote.epoch` equals `get_epoch(vote.round)`. The `get_epoch` implementation uses a reverse scan that returns the *latest* epoch whose start-round is ≤ `vote.round`:

```
pub fn get_epoch(&self, round: Round) -> Option<Epoch> {
    let epoch_start = self.epoch_starts.iter().rfind(|(_,
    start)| start <= &round);
    epoch_start.map(|(&epoch, _)| epoch)
}
```

Because the function never checks an upper bound, supplying an out-of-range value such as `Round(u64::MAX)` still yields `Some(latest_epoch)`. The vote therefore passes `verify_epoch`, even though the round is absurd.

After that check succeeds, the vote is inserted with the following:

```
self.pending_votes.entry(round).or_default();
```

This way, every distinct round number gets its own `RoundVoteState` bucket.

An attacker can send one *validly signed* vote for every round from 1 up to `u64::MAX`. None of these entries will ever be garbage collected, because they are all future rounds and `earliest_round` only moves forward.

Impact

A single malicious validator can flood memory by sending large numbers of validly signed votes for different rounds, eventually crashing honest nodes due to out-of-memory conditions. This attack does not trigger any planned multiple-vote slashing logic, because each round carries only one vote per key, making the attack stealthy. Upon a successful attack, honest nodes may exhaust memory and crash, severely degrading overall network availability and disrupting the consensus mechanism.

Recommendations

Add a future-round window in `verify_epoch` to prevent accepting rounds that are too far in the future.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `fba95788` ↗. This fix was was later incorporated into a larger commit 93b8bad3 ↗.

3.2. Denial-of-service attack via malicious block sync responses in consensus

Target	monad-consensus-state		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

At the time of writing, Monad's block-synchronization logic has error-handling flaws when processing block validation. Historically, block synchronization was only performed on blocks with a quorum certificate (QC), based on the assumption that "all blocks certified by 2f+1 validators are valid", so using `expect()` calls during validation was reasonable:

```
let author_pubkey = self
    .val_epoch_map
    .get_cert_pubkeys(&header.epoch)
    .expect("epoch should be available for blocksync'd block")
    .map
    .get(&header.author)
    .expect("blocksync'd block author should be in validator set");

let timestamp_s: u64 = (header.timestamp_ns / 1_000_000_000)
    .try_into()
    .expect("blocksync'd block timestamp > ~500B years");

let block = self
    .block_validator
    .validate(
        header,
        body,
        Some(author_pubkey),
        **tx_limit,
        **proposal_gas_limit,
        **proposal_byte_limit,
        **max_code_size,
    )
    .expect("majority extended invalid block");
```

However, the system now also synchronizes tip blocks. In the Monad consensus protocol, tip blocks are blocks that have received partial validator votes but have not yet formed a QC. When a round times out due to network delays or other reasons, the voting information from some validators is included as a tip in the time-out certificate (TC). To avoid discarding valuable blocks that have achieved partial consensus, the new round's leader will re-propose the block based on the tip information in the TC.

When a leader discovers during the `try_propose()` process that a tip block contained in the TC is missing locally, it will automatically initiate a synchronization request through `request_tip_if_missing()`. Since tip blocks have not been validated for validity through QC, validation may fail during synchronization, and `expect()` calls will cause the node to panic and crash directly.

Attackers can exploit this vulnerability by propagating TCs containing malicious tips in the network. When honest nodes attempt to synchronize these tip blocks, attackers can provide malformed or invalid block data, triggering validation failures and node crashes.

Impact

When nodes synchronize tip blocks, any validation failure will cause the node to panic and crash due to `expect()` calls, seriously affecting the network's fault tolerance and stability.

Recommendations

For all synchronized blocks, avoid proactive panics after validation failure; instead, return errors and log them to prevent node crashes due to validation failures.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `c92e615f` ↗. This fix was later incorporated into a larger commit 93b8bad3 ↗.

3.3. Time-out certificate validation allows stake amplification via duplicate NodeIds

Target	monad-consensus		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Note: This issue was known to Category Labs but remained unaddressed prior to the audit. It was later independently discovered by Zellic and fixed by Category Labs, Inc. during the assessment.

There is a serious deduplication flaw in the `verify_tc` function that allows a single validator to fake supermajority stake through repeated signatures. The problem exists on two levels.

First, during the signer-collection process, duplicate signers are blindly accepted without deduplication:

```
for t in tc.tip_rounds.iter() {
    …
    let signers = t.sigs.verify(...)?;   // Vec<NodeId<_>>
    node_ids.extend(signers);            // blindly appends, no deduplication
}
```

If the same validator injects its own signature into multiple `HighTipRoundSigColTuples`, that `NodeId` is appended multiple times to `node_ids`.

Second, the vote-counting routine does no deduplication in release builds:

```
fn has_super_majority_votes(&self, addrs: &[NodeId<_>]) -> bool {
    let mut duplicates = HashSet::new();
    …
    debug_assert!(duplicates.insert(addr)); // stripped in release
    voter_stake += *v;                      // stake added every time
}
```

Only in debug mode does `debug_assert!` execute; in a release binary, `duplicates.insert(addr)` is never called, so the same `NodeId`'s stake is accumulated repeatedly.

By repeating its own signature, a single adversarial validator can inflate `voter_stake` without limit; once it reaches `2/3 × total_stake + 1`, the node can forge a valid time-out certificate (TC).

Impact

This vulnerability will result in permanent denial of service. Consider an attacker A sending a `TimeoutMessage` to honest nodes in round 21:

```
{
    tminfo: {
        epoch: 1,
        round: 1000001, // forged extremely large value, but 1 greater than
    last_round_tc round to ensure validity
        high_qc_round: 20,
    },
    timeout_signature: sig_A,
    high_extend: QC(20),
    last_round_tc: {
        epoch: 1,
        round: 1000000, // forged extremely large value
        tip_rounds: [{high_qc_round: 20, sigs: bls_sig_A}, {high_qc_round:
    20, sigs: bls_sig_A}...]
        high_extend: QC(20)
    }
}
```

The `TimeoutMessage` validate function

```
pub fn validate<VTF, VT, LT>(
    self,
    epoch_manager: &EpochManager,
    val_epoch_map: &ValidatorsEpochMapping<VTF, SCT>,
    election: &LT,
) -> Result<Validated<TimeoutMessage<ST, SCT, EPT>>, Error>
where
    VTF: ValidatorSetTypeFactory<ValidatorSetType = VT>,
    VT: ValidatorSetType<NodeIdPubKey = SCT::NodeIdPubKey>,
    LT: LeaderElection<NodeIdPubKey = SCT::NodeIdPubKey>,
{
    // If there's a timeout certificate from the previous round, validate it
    if let Some(tc) = &self.obj.last_round_tc {
        // Verify the timeout certificate
        verify_tc(
            &|epoch, round| {
                epoch_to_validators(epoch_manager, val_epoch_map, election,
    epoch, round)
            },
            tc,
        )?;
        // last_round_tc must come from the previous round
        if self.obj.tminfo.round != tc.round + Round(1) {
            return Err(Error::NotWellFormed);
        }
    }
    // Verify high extend
    verify_high_extend(
        &|epoch, round| {
            epoch_to_validators(epoch_manager, val_epoch_map, election, epoch,
    round)
        },
        &self.obj.high_extend.clone().into(),
    )?;

    // Check if timeout message is well-formed
    self.well_formed_timeout()?;
    // Verify epoch
    self.verify_epoch(epoch_manager)?;

    // Return validated timeout message
    Ok(Validated { message: self })
}
```

calls `verify_tc` to check if there is a TC from the previous round. This function first checks if the 2/3 weight requirement is met then validates the `high_extend` within it. Weight validation can be bypassed by repeatedly injecting its own signature into several `HighTipRoundSigColTuples`. Therefore, we can issue a completely forged TC. `high_extend` validation simply requires using a legitimate `high_extend`, which is QC(20) in this case.

Next, it checks `self.obj.tminfo.round != tc.round + Round(1)`. Since both values are under the attacker's control, this condition can be easily bypassed by setting them to `1000001` and `1000000`, respectively. Then it calls `verify_high_extend` to check the `high_extend` again, which can also be passed by providing a legitimate `high_extend`. The `well_formed_timeout` check only requires the existence of `self.obj.last_round_tc`. The `verify_epoch` check can be bypassed because regardless of how large the round number is, it only retrieves the latest epoch.

The problem occurs later in `handle_timeout_message`:

```
pub fn handle_timeout_message(
    &mut self,
    author: NodeId<SCT::NodeIdPubKey>,
    timeout: TimeoutMessage<ST, SCT, EPT>,
) -> Vec<ConsensusCommand<ST, SCT, EPT, BPT, SBT>> {
    ...
    if let Some(last_round_tc) = timeout.last_round_tc.as_ref() {
        info!(?last_round_tc, "advance round from remote TC");
        // Update remote timeout message with TC metric counter
        self.metrics.consensus_events.remote_timeout_msg_with_tc += 1;
        // Process TC certificate to advance round
        let advance_round_cmds = self
            .consensus
            .pacemaker
            .process_certificate(
                self.metrics,
                self.epoch_manager,
                &mut self.consensus.safety,
                RoundCertificate::Tc(last_round_tc.clone()),
            )
    ...
```

The issue lies in how it processes the `last_round_tc`. The `process_certificate` function

```
#[must_use]
pub fn process_certificate(
    &mut self,
    metrics: &mut Metrics,
    epoch_manager: &EpochManager,
    safety: &mut Safety<ST, SCT, EPT>,
    certificate: RoundCertificate<ST, SCT, EPT>,
) -> Vec<PacemakerCommand<ST, SCT, EPT>> {
    // If certificate round is not higher than current high certificate round,
    ignore it
    if certificate.round() <= self.high_certificate.round() {
        return Default::default();
    }
    ...
    // Let safety module process the certificate
    safety.process_certificate(&certificate);
    // Update high certificate
    self.high_certificate = certificate;
}
```

anchors the current round to this forged TC certificate through `self.high_certificate = certificate`, causing `get_current_round` to return an extremely large current round number. As a result, it can never process any messages again, leading to persistent denial of service.

Recommendations

Always deduplicate inside `has_super_majority_votes`; move `duplicates.insert(addr)` out of the `debug_assert!`.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `69df9e4f` ↗.

3.4. Zero-length header request triggers remote denial-of-service panic

Target	monad-blocksync		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `handle_peer_request` function has a serious vulnerability when processing externally incoming `BlockSync` requests. If an attacker sends a `BlockSyncRequestMessage::Headers` with `num_blocks` set to 0, the node walks into the following logic trap:

```
pub fn handle_peer_request(
    &mut self,
    sender: NodeId<SCT::NodeIdPubKey>,
    request: BlockSyncRequestMessage,
) -> Vec<BlockSyncCommand<ST, SCT, EPT>> {
    match request {
        BlockSyncRequestMessage::Headers(block_range) => {
            let cached_blocks = match self.block_cache {
                BlockCache::BlockTree(blocktree) => blocktree
                    .get_parent_block_chain(&block_range.last_block_id)
                    .into_iter()
                    .map(|block| block.header().clone())
                    .rev()
                    .take(block_range.num_blocks.0 as usize)  // take(0)
                    .rev()
                    .collect_vec(),
                BlockCache::BlockBuffer(_) => Vec::new(),
            };

            // check if all blocks are cached
            if cached_blocks.len() == block_range.num_blocks.0 as usize {  //
    0 == 0
                assert_eq!(
                    Some(block_range.last_block_id),
                    cached_blocks.last().map(|block| block.get_id())  // None
                );
```

The attack flow is as follows:

1. The code calls `take(block_range.num_blocks.0 as usize)`. With `num_blocks == 0`, it performs `take(0)`, so `cached_blocks` becomes an *empty vector*.
2. It immediately checks `cached_blocks.len() == block_range.num_blocks.0 as usize`. Because both sides are `0`, the branch is taken — the program now believes "all requested blocks are already cached".
3. Inside that branch, it executes the assertion `cached_blocks.last()` returns `None`, while the left-hand side of the assertion is `Some(last_block_id)`. The assertion therefore fails and panics, crashing the whole process.

Because no prior check rejects a zero-length range, *any* reachable peer can trigger a panic with a single packet, yielding a reliable remote denial of service.

Impact

Any malicious node can crash target nodes by sending a single malicious `BlockSyncRequestMessage::Headers` packet containing `num_blocks = 0`. This attack has the following characteristics:

- Extremely low attack cost, requiring only a single packet
- Severe impact, causing the entire node process to crash
- Reliability, with a 100% success rate
- No special privileges required (any reachable peer can execute it)

In a network environment, a single malicious node can systematically attack all honest nodes, causing the entire network to be paralyzed and consensus to be unable to continue.

The following proof of concept (POC) demonstrates the real-world impact of this vulnerability. This POC is run after starting four nodes with `flexnet ./topologies/4nodes-full.json`. During vote processing in round 20, a malicious node sends malicious block-sync messages to the other three nodes:

```
diff --git a/monad-consensus-state/src/lib.rs
    b/monad-consensus-state/src/lib.rs
index 784344af..a5d84c7d 100644
-- a/monad-consensus-state/src/lib.rs
++ b/monad-consensus-state/src/lib.rs
@@ -588,6 +588,24 @@ where
         author: NodeId<SCT::NodeIdPubKey>,
         vote_msg: VoteMessage<SCT>,
     ) -> Vec<ConsensusCommand<ST, SCT, EPT, BPT, SBT>> {
        let mut cmds = Vec::new();

        if vote_msg.vote.round == Round(20) {
            let block_range = BlockRange {
                last_block_id: vote_msg.vote.id,
                num_blocks: SeqNum(0),  // Zero-length attack payload
            };

            let blocksync_cmd = ConsensusCommand::RequestSync(block_range);
            cmds.push(blocksync_cmd);
        }
```

These are the results from the three honest nodes: all three panicked, leaving only the malicious node online. The network is stuck at round 20 and can no longer produce blocks, resulting in a complete denial of service:

```
thread 'main' panicked at
    /usr/src/monad-bft/monad-blocksync/src/blocksync.rs:338:21:
assertion `left == right` failed
  left: Some(496b..a135)
 right: None
stack backtrace:
   0: __rustc::rust_begin_unwind
             at
    /rustc/17067e9ac6d7ecb70e50f92c1944e545188d2359/library/std/src/...
        panicking.rs:697:5
   1: core::panicking::panic_fmt
             at
    /rustc/17067e9ac6d7ecb70e50f92c1944e545188d2359/library/core/src/...
        panicking.rs:75:14
   2: core::panicking::assert_failed_inner
             at
    /rustc/17067e9ac6d7ecb70e50f92c1944e545188d2359/library/core/src/...
        panicking.rs:425:17
   3: core::panicking::assert_failed
             at
    /rustc/17067e9ac6d7ecb70e50f92c1944e545188d2359/library/core/src/...
        panicking.rs:380:5
   4: monad_blocksync::blocksync::BlockSyncWrapper<ST,SCT,EPT,BPT,SBT,...
        VTF>::handle_peer_request
             at ./monad-blocksync/src/blocksync.rs:338:21
   5:
    monad_state::blocksync::BlockSyncChildState<ST,SCT,EPT,BPT,SBT,VTF,LT,BVT,...
        CCT,CRT>::update
             at ./monad-state/src/blocksync.rs:102:17
   6: monad_state::MonadState<ST,SCT,EPT,BPT,SBT,VTF,LT,BVT,CCT,CRT>::update
             at ./monad-state/src/lib.rs:893:39
```

This demonstrates that the vulnerability can effectively bring down an entire consensus network with minimal effort from a single malicious actor.

Recommendations

Reject `num_blocks == 0` requests at input boundaries, or turn the `assert_eq!` into a recoverable comparison so that malformed requests are ignored, instead of killing the node.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `73d76e74` ↗.

3.5. Integer Overflow in Transaction Gas Cost Calculation

Target	monad-eth-block-policy		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Note: This issue was also discovered independently and fixed by Category Labs, Inc. during the assessment.

An integer overflow vulnerability exists in the consensus layer's `compute_txn_max_value` function. When calculating the maximum transaction cost, integer overflow occurs when `max_fee_per_gas` is an extremely large u128 value:

```
pub fn compute_txn_max_value(txn: &TxEnvelope) -> U256 {
    let txn_value = U256::try_from(txn.value()).unwrap();
    let gas_cost = U256::from(txn.gas_limit() as u128 *
    txn.max_fee_per_gas());
    txn_value.saturating_add(gas_cost)
}
```

The data type conversion in the calculation is `U256((u64 as u128) * u128)`. When `max_fee_per_gas` approaches the u128 maximum value, the multiplication operation causes integer overflow, resulting in wrap-around behavior in Rust's release mode, producing an erroneously small `gas_cost` value that allows the transaction to pass balance validation in the consensus layer.

However, the execution layer uses a different calculation method:

```
inline intx::uint512 max_gas_cost(uint64_t const gas_limit,
    uint256_t max_fee_per_gas) noexcept {
    return intx::umul(uint256_t{gas_limit}, max_fee_per_gas);
}
```

The execution layer uses 512-bit integer calculation, which does not overflow and produces the correct large gas cost value, leading to insufficient balance validation failure.

This calculation discrepancy creates serious security issues under Monad's delayed validation mechanism. During `validate_block_body`, the consensus layer only performs `static_validate_transaction` static validation without synchronously executing transactions. Block headers store `delayed_execution_results` rather than the state root of transactions within the current block, adopting an optimistic commit strategy. `check_coherency(k)` only validates execution results at height `k - execution_delay`, meaning block k may be marked as coherent before its own execution results are generated. If `execution_delay` is set larger than the HotStuff three-chain commit depth (typically 2), blocks may be finalized before complete validation. When execution failures are discovered later, rollback becomes impossible, compromising consensus security.

Impact

An attacker can construct a transaction with a `max_fee_per_gas` value close to the u128 maximum, which passes balance validation in the consensus layer due to integer overflow, allowing the block to be marked as coherent and potentially committed. When the execution layer processes this transaction, it calculates the correct large gas cost, causing insufficient balance failure. Due to Monad's delayed validation mechanism, the block may already be non-rollback-able at this point, leading to consensus forks and network stability issues.

Recommendations

Use checked arithmetic or larger integer types in the consensus layer to prevent overflow and ensure consistency with execution layer calculation logic.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `f6c484a6` ↗.

3.6. Denial of service in SignerMap RLP decode

Target	monad-bls		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The vulnerability exists in the `SignerMap::decode` function where `num_bits` is read without any upper-bound validation. The decoding process enters a `while bitvec.len() < num_bits` loop that executes `num_bits` times to construct a BitVec:

```
fn decode(buf: &mut &[u8]) -> alloy_rlp::Result<Self> {
    let mut payload = alloy_rlp::Header::decode_bytes(buf, true)?;
    let num_bits: usize = <u32 as Decodable>::decode(&mut payload)?
    as usize;  // No bounds check
    let num_bytes = num_bits.div_ceil(8);

    let decoded_bytes = alloy_rlp::Header::decode_bytes(&mut payload, false)?;
    if decoded_bytes.len() != num_bytes {
        return Err(alloy_rlp::Error::Custom("wrong number of bytes in
    bitvec"));
    }

    let mut bitvec = BitVec::with_capacity(num_bits);
    while bitvec.len() < num_bits {  // Vulnerable loop
        let byte_idx_from_back = bitvec.len() / 8;
        let bit_idx_from_back = bitvec.len() % 8;

        let byte = decoded_bytes[decoded_bytes.len() - 1 -
    byte_idx_from_back];
        let bit = (byte >> bit_idx_from_back) & 1 == 1;
        bitvec.push(bit);
    }
    bitvec.reverse();

    Ok(SignerMap(bitvec))
}
```

With current TCP message limits of 1GB, an attacker can set `num_bits` to 8GB (eight billion bits), causing approximately eight billion loop iterations, and effectively cause a denial of service to the target node.

Impact

The vulnerability can be triggered through multiple consensus-protocol paths. In the current 1GB TCP limit scenario, if an attacker sets `num_bits` to 8GB when encoding locally and sends a 1GB message, the decoding `while bitvec.len() < num_bits` loop will execute approximately eight billion iterations, causing target nodes to enter prolonged loops during decoding, resulting in denial of service.

Even in future scenarios where TCP message size is reduced to 3MB, this will still cause 0.5–1 second processing delays. Malicious nodes can continuously send multiple time-out messages, all of which will be processed. Given that the expected time-out duration per round (returned by `get_round_timer`) is approximately 2.1 seconds, a single malicious node can force other nodes into time-out loops. Therefore, even with TCP message-size mitigation, this vulnerability cannot be prevented from being exploited.

The following test confirms the attack feasibility under 3MB TCP limits:

```
#[test]
fn test_signer_map_rlp_3mb() {
    use bitvec::prelude::*;

    const NUM_BITS: usize = 3 * 1024 * 1024 * 8;  // 3MB worth of bits

    let big_map1 = SignerMap(BitVec::<u8, Lsb0>::repeat(true, NUM_BITS));
    let encoded1 = alloy_rlp::encode(big_map1.clone());
    let mut input1 = encoded1.as_slice();
    let decoded1 = <SignerMap>::decode(&mut input1).unwrap();

    assert_eq!(big_map1, decoded1);
}
```

The test completes but takes 1.22 seconds, demonstrating that message-size limits alone are insufficient.

Recommendations

Add bounds checking for `num_bits` in the decode function, and further restrict network layer packet sizes to reduce the attack surface.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `ed52e335` ↗.

3.7. Unbounded `tip_rounds` processing: `TimeoutCertificate` denial of service

Target	monad-consensus		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The vulnerability exists in the `TimeoutCertificate` processing logic where the `tip_rounds` vector lacks size validation. The structure allows unlimited `HighTipRoundSigColTuple` entries, each triggering expensive BLS signature verification without caching or deduplication:

```
pub struct TimeoutCertificate<ST, SCT, EPT> {
    pub epoch: Epoch,
    pub round: Round,
    pub tip_rounds: Vec<HighTipRoundSigColTuple<SCT>>,  // No size limit
    pub high_extend: HighExtend<ST, SCT, EPT>,
}

pub struct HighTipRoundSigColTuple<SCT> {
    pub high_qc_round: Round,
    pub high_tip_round: Round,
    pub sigs: SCT,
}
```

The vulnerable processing loop performs BLS verification for each tuple without bounds checking:

```
for t in tc.tip_rounds.iter() {             // tip_rounds has no upper bound
    let signers = t.sigs
        .verify::<signing_domain::Timeout>(validator_mapping, msg.as_ref())
        .map_err(|_| Error::InvalidSignature)?;  // Expensive BLS operation

    node_ids.extend(signers);
}
```

Two key issues enable the attack: 1) the unbounded `tip_rounds` length allowing thousands of entries and 2) no verification caching, causing repeated expensive BLS operations.

Impact

An attacker can craft a 3MiB `TimeoutCertificate` message containing approximately 29,000–32,000 `HighTipRoundSigColTuple` entries. Given that each signature collection (`sigs`) is 97–109 bytes (`N ≤ 100` validators), this allows for massive tuple counts within the message-size limit. With each BLS signature verification taking 0.5–1ms, processing time ranges from 14.5 seconds (optimistic) to 32 seconds (typical). This blocks single-threaded consensus logic for 15–32 seconds per malicious message.

Through raptor UDP transmission, a complete consensus message can contain up to 65,535 packets, with each UDP datagram capped at the default MTU of 1480 bytes, making the largest consensus message a node can transmit approximately `65535 * 1480 = 97MB`, further amplifying the potential impact of the attack.

```rust
#[test]
fn test_timeout_certificate_dos() {
    let mut tip_rounds = Vec::new();

    // Create 30,000 malicious tip_rounds entries
    for i in 0..30000 {
        tip_rounds.push(HighTipRoundSigColTuple {
            high_qc_round: Round(i),
            high_tip_round: Round(i + 1),
            sigs: create_valid_signature_collection(),
        });
    }

    let malicious_tc = TimeoutCertificate {
        epoch: Epoch(1),
        round: Round(100),
        tip_rounds,
        high_extend: create_valid_high_extend(),
    };

    let start = std::time::Instant::now();
    let result = process_timeout_certificate(malicious_tc);
    let duration = start.elapsed();

    println!("Processing time: {:?}", duration);
}
```

Recommendations

Set a strict upper bound for the `tip_rounds` length, and reject oversized messages immediately.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `9da31b19` ↗.

3.8. Signature verification exhaustion via malicious looping in delayed verification

Target	monad-consensus		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The vulnerability exists in `VoteState::process_vote` implementation where the system uses delayed verification mechanism: after a vote arrives, its signature is not verified immediately. Instead, the `(NodeId, Signature)` pair is first stored in the corresponding vote bucket; only when the total stake in that bucket reaches ≥ 2f + 1 does the code call `SignatureCollection::new()` to batch-verify every signature.

```
// Loop to check if super majority votes are reached
while validators

    .has_super_majority_votes(&round_pending_votes.keys().copied().collect::<Vec<_>>())
{
    assert!(round >= self.earliest_round);
    let vote_enc = alloy_rlp::encode(vote);
    match SCT::new::<signing_domain::Vote>(
        round_pending_votes
            .iter()
            .map(|(node, signature)| (*node, *signature)),
        validator_mapping,
        vote_enc.as_ref(),
    ) {
        Ok(sigcol) => {
            let qc = QuorumCertificate::<SCT>::new(vote, sigcol);
            self.earliest_round = round + Round(1);
            return (Some(qc), ret_commands);
        }
        Err(SignatureCollectionError::InvalidSignaturesCreate(invalid_sigs))
    => {
            let cmds = Self::handle_invalid_vote(round_pending_votes,
    invalid_sigs);
            ret_commands.extend(cmds);
        }
        Err(_) => {
            unreachable!("InvalidSignaturesCreate is only expected error from
    creating SC");
```

```
            }
        }
    }
```

The core issue is that the system uses delayed verification mechanism, first storing votes in buckets without immediately verifying signature validity. Only when the bucket's weight reaches super majority (≥2f+1) does it trigger batch verification. When `SignatureCollection::new()` returns `InvalidSignaturesCreate` error, the system removes invalid signatures but continues to loop and check if super majority condition is still satisfied, with no limit on the number of repeated verifications within the same round.

An attacker first injects a forged signature into a vote bucket, then waits for honest validators to naturally accumulate votes until the bucket's weight reaches ≥2f+1. This triggers batch verification which will fail due to the invalid signature. The system then removes the malicious entry, but since other honest votes remain, the weight may still satisfy the super majority condition, allowing the attacker to immediately inject another forged signature, triggering verification again and forming an infinite loop.

In each loop iteration, the system executes expensive operations including RLP encoding vote data (`alloy_rlp::encode(vote)`), batch BLS signature verification (`SignatureCollection::new()`), and traversing and processing all pending votes.

Impact

A malicious validator can cause temporary CPU exhaustion by repeatedly injecting invalid signatures when vote buckets reach super majority, forcing expensive batch verification operations in a loop, though the attack's effectiveness is limited unless the attacker controls sufficient stake weight.

Recommendations

Set an upper limit on the number of batch verifications per round to prevent infinite loops, consider performing basic signature verification when storing votes rather than completely delaying to batch processing, track the frequency of each node sending invalid signatures and implement temporary restrictions on nodes that frequently send invalid signatures, and cache verified signatures to avoid repeatedly verifying the same signatures.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit 48c756c9 ↗.

3.9. The PeerEntry RLP-decoding cursor misalignment

Target	monad-executor-glue		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Proper RLP decoding requires consistent use of the same payload pointer throughout the process. As each field is decoded, the payload cursor should automatically advance to ensure sequential consistency.

In this implementation, only the `pubkey` field uses `&mut payload`, while subsequent fields (`s`, `signature`, `record_seq_num`) use `buf` instead:

```rust
impl<ST: CertificateSignatureRecoverable> Decodable for PeerEntry<ST> {
    fn decode(buf: &mut &[u8]) -> alloy_rlp::Result<Self> {
        let mut payload = alloy_rlp::Header::decode_bytes(buf, true)?;

        let pubkey = CertificateSignaturePubKey::<ST>::decode(&mut payload)?;
        let s = <String as Decodable>::decode(buf)?; // Issue here
        let addr = s
            .parse::<SocketAddrV4>()
            .map_err(|_| alloy_rlp::Error::Custom("invalid SocketAddrV4"))?;
        let signature = ST::decode(buf)?; // Issue here
        let record_seq_num = u64::decode(buf)?; // Issue here

        Ok(Self {
            pubkey,
            addr,
            signature,
            record_seq_num,
        })
    }
}
```

This causes each decoding operation to start from the beginning of the original byte stream, leading to cursor corruption. The result is that `pubkey` decodes correctly, but subsequent fields either decode incorrect data or fail with errors such as `InputTooShort`.

Impact

All functionality dependent on `PeerEntry` deserialization will malfunction. This severely affects internode communication and network functionality, potentially causing nodes to fail to correctly identify or connect to other peers, thereby undermining network connectivity and stability.

The following test case can reproduce this issue:

```
#[test]
fn peer_entry_rlp_roundtrip() {
    use monad_crypto::NopSignature;
    use monad_crypto::certificate_signature::CertificateSignaturePubKey;
    use std::net::SocketAddrV4;

    let pubkey = CertificateSignaturePubKey::<NopSignature>::from_bytes(&[1u8;
    32]).unwrap();
    let addr: SocketAddrV4 = "127.0.0.1:12345".parse().unwrap();
    let signature = NopSignature { pubkey, id: 12345678 };
    let record_seq_num = 42u64;
    let entry = PeerEntry {
        pubkey,
        addr,
        signature,
        record_seq_num,
    };
    let encoded = alloy_rlp::encode(&entry);
    let decoded: PeerEntry<NopSignature>
    = alloy_rlp::decode_exact(&encoded).unwrap();
    assert_eq!(entry, decoded);
}
```

Recommendations

Change all field decoding operations to use `&mut payload` consistently to ensure the cursor advances correctly.

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `cc20e356` ↗.

3.10. Generic constant NUM is ignored in compute_upcoming_leader_round_pairs implementation

Target	monad-state		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `compute_upcoming_leader_round_pairs` has a logical flaw. This function defines a generic constant parameter `NUM` to control the number of returned leader-round pairs, but in its actual implementation, it does not use this generic parameter `NUM`. Instead, it uses a hardcoded constant `NUM_LEADERS_FORWARD_TXS`:

```
fn compute_upcoming_leader_round_pairs<
    const INCLUDE_CURRENT_ROUND: bool,
    const SKIP_SELF: bool,
    const NUM: usize,
>(
    &self,
) -> Vec<(NodeId<CertificateSignaturePubKey<ST>>, Round)> {
    let ConsensusMode::Live(mode) = &self.consensus else {
        return Vec::default();
    };

    (mode.get_current_round().0 + (if INCLUDE_CURRENT_ROUND { 0 } else {
    1 })..)
        .take(NUM_LEADERS_FORWARD_TXS)  // Should use NUM instead of
    NUM_LEADERS_FORWARD_TXS
```

This function has two call sites that expect to obtain different numbers of leader rounds by specifying different `NUM` values, but they actually both receive the same number (`NUM_LEADERS_FORWARD_TXS`) of results.

Impact

The generic parameter `NUM` specified by external callers is completely ineffective, causing callers to potentially receive data inconsistent with their expectations.

Recommendations

Replace the hardcoded constant in the function implementation with the generic parameter (i.e., change `.take(NUM_LEADERS_FORWARD_TXS)` to `.take(NUM)`).

Remediation

This issue has been acknowledged by Category Labs, Inc., and a fix was implemented in commit `2f546cff` ↗.

3.11. RLP `Option` field double-encoding causes decoding failure

Target	monad-executor-glue		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the Monad project's `MempoolEvent::Proposal` RLP-encoding implementation, the `Option`-type fields `last_round_tc` and `fresh_proposal_certificate` are first encoded into an intermediate `BytesMut` buffer and then inserted as byte strings into the outer RLP list. This approach causes already encoded RLP fragments to be wrapped again as strings, resulting in double-encoding.

In the first encoding, `encode_list` generates RLP fragments, encoded as `[0xc1, 0x01]`, where `0xc1` indicates the list prefix:

```
let mut tc_buf = BytesMut::new();
match last_round_tc {
    None => {
        let enc: [&dyn Encodable; 1] = [&1u8];
        encode_list::<_, dyn Encodable>(&enc, &mut tc_buf);
    }
    Some(tc) => {
        let enc: [&dyn Encodable; 2] = [&2u8, &tc];
        encode_list::<_, dyn Encodable>(&enc, &mut tc_buf);
    }
}
```

In the second encoding, the outer `encode_list` treats `&tc_buf` as a byte string, adding string prefix `0x82`, resulting in `0x82 0xc1 0x01`:

```
let enc: [&dyn Encodable; 11] = [
    &1u8, epoch, round, seq_num, high_qc, timestamp_ns, round_signature,
    delayed_execution_results, proposed_execution_inputs,
    &tc_buf, // This gets wrapped as a byte string again
    // ...
];
encode_list::<_, dyn Encodable>(&enc, out);
```

During decoding, the RLP header expects a list (`0xc1`) but encounters a string prefix (`0x82`), causing an `UnexpectedString` error and decode failure.

Impact

This vulnerability causes `MempoolEvent::Proposal` RLP-decoding failure. However, the decoder codepath is never called in production environments. The encoder is only used for writing the debugging event `WAL`, while the decoder is only used in the `wal2json` debugging tool, which has not been used in recent months (as of the time of writing). Therefore, despite the technical flaw, the actual impact on production systems is minimal.

The following test case can reproduce this issue:

```
#[test]
fn mempoolevent_rlp_roundtrip() {
    use monad_crypto::NopSignature;
    use monad_consensus_types::payload::RoundSignature;
    use monad_consensus_types::block::{ProposedExecutionInputs,
    MockExecutionProtocol};
    use monad_multi_sig::MultiSig;
    use monad_types::{Epoch, Round, SeqNum, NodeId};
    use bytes::Bytes;

    type ST = NopSignature;
    type SCT = MultiSig<NopSignature>;
    type EPT = MockExecutionProtocol;

    // Construct Proposal
    let pubkey_bytes = [1u8; 32];
    let mut key_bytes = pubkey_bytes;
    let keypair = monad_crypto::NopKeyPair::from_bytes(&mut
    key_bytes).unwrap();
    let proposal = MempoolEvent::<ST, SCT, EPT>::Proposal {
        epoch: Epoch(1),
        round: Round(2),
        seq_num: SeqNum(3),
        high_qc:
    monad_consensus_types::quorum_certificate::QuorumCertificate::<SCT>
        ::genesis_qc(),
        timestamp_ns: 123456789,
        round_signature: RoundSignature::new(Round(2), &keypair),
        delayed_execution_results: vec![],
        proposed_execution_inputs: ProposedExecutionInputs::<EPT> {
            header:
    monad_consensus_types::block::MockExecutionProposedHeader::default(),
            body: monad_consensus_types::block::MockExecutionBody::default(),
        },
        last_round_tc: None,
        fresh_proposal_certificate: None,
    };
    let encoded = alloy_rlp::encode(&proposal);
    let decoded: MempoolEvent<ST, SCT, EPT>
    = alloy_rlp::decode_exact(&encoded).unwrap();
    assert_eq!(proposal, decoded);
    // Construct ForwardedTxs
    let pubkey_bytes = [1u8; 32];
    let mut key_bytes = pubkey_bytes;
    let keypair = monad_crypto::NopKeyPair::from_bytes(&mut
    key_bytes).unwrap();
    let pubkey = keypair.pubkey();
    let sender = NodeId::new(pubkey);
    let forwarded = MempoolEvent::<ST, SCT, EPT>::ForwardedTxs {
        sender,
        txs: vec![Bytes::from_static(b"tx1"), Bytes::from_static(b"tx2")],
    };
    let encoded = alloy_rlp::encode(&forwarded);
    let decoded: MempoolEvent<ST, SCT, EPT>
    = alloy_rlp::decode_exact(&encoded).unwrap();
    assert_eq!(forwarded, decoded);

    // Construct ForwardTxs variant
    let forward = MempoolEvent::<ST, SCT,
    EPT>::ForwardTxs(vec![Bytes::from_static(b"tx3")]);
    let encoded = alloy_rlp::encode(&forward);
    let decoded: MempoolEvent<ST, SCT, EPT>
    = alloy_rlp::decode_exact(&encoded).unwrap();
    assert_eq!(forward, decoded);
}
```

Recommendations

Do not insert already encoded RLP fragments as byte strings into the outer list. Instead, directly insert the `Option`'s tag and content as RLP list elements, avoiding intermediate buffers.

Remediation

TBD

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Trust Model Risk in State Sync

The current state sync implementation poses an over-reliance risk on pre-configured trusted nodes. The current implementation assumes that pre-configured trusted nodes are always secure and reliable, lacking sufficient runtime validation of messages from these nodes. This design exposes serious system vulnerabilities when trusted nodes are compromised.

Consider this scenario: an attacker successfully compromises a pre-configured trusted node. Since the system completely trusts this node, malicious messages sent by the attacker can easily bypass identity verification checks. Once authentication passes, the system directly processes responses from trusted nodes without any additional sanity checks.

```
// Direct processing after trusted node verification
if !self.state_sync_peers.iter().any(|trusted| trusted == &from) {
    return; // Compromised trusted node bypasses this check
}
...
let replaced = self.responses.insert(response.response_index, response);
assert!(replaced.is_none(), "server sent duplicate response_index");
```

Here's an example of a risky code path: if a compromised trusted node sends two responses with the same `response_index`, when the second response arrives, it triggers an `assert!` failure, causing the entire node process to crash.

This design flaw brings multifaceted risks: a single compromised trusted node can cause more nodes to crash as well, and the system lacks detection and response mechanisms for abnormal behavior from trusted nodes.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Monad Consensus

Monad Consensus implements a round-driven state-machine replication protocol based on HotStuff, ensuring both safety (no forks) and liveness (continuous progress) of the blockchain state machine in a Byzantine fault tolerant (BFT) environment. The module adopts linear communication and pipelined design where designated leaders propose new blocks each round, validators send votes to the next round's leader after validating proposals, and the next round's leader aggregates votes to form quorum certificates (QCs). When two consecutive rounds' proposals obtain QCs, the earlier proposal can be committed as a final decision.

Under normal operation, leaders propose in sequence, validators vote to form consecutive QC chains, and the protocol progresses efficiently. In exceptional situations such as leader failures or network issues, the module quickly switches leaders and enters new rounds through time-out messages and round-recovery mechanisms, thereby ensuring protocol liveness.

The entire protocol is driven by five types of core consensus messages, implementing the complete state-machine replication process from proposal generation to final commitment.

Five types of consensus messages

The consensus state machine progresses primarily around five message types: `Proposal`, `Vote`, `Timeout`, `RoundRecovery`, and `NoEndorsement`. Each message type has its specific purpose and triggering conditions. Below we explain their functions, transmission timing, validation logic, and impact on the state machine after passing validation.

1. Message type: `Proposal`

This section outlines the structure, basic functions, and life cycle of `Proposal` messages.

Message structure and basic functions

The `Proposal` messages carry block proposals from leaders and serve as the starting point for the consensus protocol. Proposals typically include the current round number, proposed block data, QC (or time-out certificate, TC) of the preceding block, and the leader's signature on the proposal. The core function is to propagate new block candidates to all validators, initiating a new round of consensus voting. Proposal messages ensure legal linking with previously authenticated blocks or handling of failed rounds: if the previous round formed a QC, the new `Proposal` references that QC to link the blockchain; if the previous round failed and a TC was collected, the `Proposal` references the TC as safety evidence.

Message life cycle

Creation and propagation. At the beginning of each round, the node elected as leader collects pending transactions and packages them into a new block, constructing a `Proposal` message. The triggering condition enters a new consensus round with this node elected as the round's leader, when Pacemaker drives this node to create and send a `Proposal`. After creating the `Proposal`, the leader node broadcasts it to all validators. Under normal circumstances, each round has only one unique leader sending `Proposal`s, with other nodes as receivers, allowing all validators to participate in subsequent voting by receiving the `Proposal`. When the network is healthy and the leader is honest, most rounds begin with normal `Proposal` broadcasts.

Reception and validation. After validator nodes receive `Proposal` messages, they verify their legality through the signature module, including the following validation steps:

- *Message-integrity verification.* Using EpochManager's current epoch validator public keys (`val_epoch_map`), verify that the `Proposal` message's digital signature was issued by the declared leader, ensuring message integrity and trustworthiness. Through LeaderElection, calculate the expected leader for the current round and check whether the `Proposal` message sender is indeed the legitimate leader for this round.
- *State-consistency verification.* Check whether the QC/TC carried by the `Proposal` is legal and consistent with the current state. For example, if it contains the previous round's QC, verify the QC signature and round matching, ensuring the proposed block correctly extends the block specified by that QC; if it contains the previous round's TC, extract the highest QC information from the TC and verify whether the new proposal extends the block pointed to by the highest QC noted in the TC (ensuring no higher legitimate branches are skipped).
- *Round and block validation.* Check the `Proposal`'s round field. Its round must not be higher than the current round; otherwise, it is considered a future proposal and discarded. Finally, nodes verify the block carried in the proposal message through `validate_block`. Only after passing these validations is the `Proposal` considered valid.

State updates and subsequent operations. When a node accepts a legal `Proposal`, it performs the following operations.

1. First, temporarily store the proposed block in the local block tree or cache, updating local QC/locking state (for example, if the `Proposal`'s attached QC is newer than the current highest QC, update the highest QC) to reflect the latest network progress.

2. Then, the node needs to decide whether to vote on this proposal through multiple layers of checks: timestamp-validity verification, block-consistency checks, and safe voting checks (ensuring the round is greater than the historical highest voting round).

3. After passing these checks, create a `Vote` message and record it in the Safety module.

Then decide whether to send immediately or delay sending the vote to the next round's leader based on voting scheduling status.

Thus, the `Proposal` message triggers validators' local state updates and subsequent `Vote` generation. If verification fails for any reason (such as signature errors, the sender not being a legitimate leader, an invalid timestamp, block inconsistency, etc.), the node will reject the `Proposal` and not vote, thereby ensuring protocol safety.

2. Message type: `Vote`

This section outlines the structure, basic functions, and life cycle of `Vote` messages.

Message structure and basic functions

The `Vote` messages represent validators' endorsement of a specific `Proposal`, carrying vote content and voting signatures. The core function is to allow leaders to collect majority node agreement on proposals, thereby forming QCs that provide BFT confirmation proof for the block. Votes include the proposed block hash, the proposal's round, and so on used for subsequent QC generation. Vote messages are typically not broadcast like proposals but are sent to the next round's leader. This way, the next round's leader aggregates votes to form QCs, avoiding the overhead of broadcasting every vote across the network and reducing message complexity.

Message life cycle

Creation and propagation. The triggering condition is when validators successfully validate a `Proposal` and decide to support the proposal, immediately generating a `Vote` message. Under normal circumstances, validators trigger a vote each time they receive a valid `Proposal`. Validator nodes check voting conditions after processing the `Proposal` and, if satisfied, use their private key to sign and generate a `Vote` message. Validators send the `Vote` to the next round's leader node.

Reception and validation. When the new round's leader receives `Vote` messages from other nodes, validation includes the following:

- *Message-integrity verification.* Using EpochManager and `val_epoch_map` to verify the `Vote` message's digital signature, ensure the message was issued by the sender.
- *Round check.* Check whether the voting round is not less than the earliest round (`earliest_round`); votes from expired rounds are directly discarded.
- *Epoch verification.* Check whether the epoch corresponding to the voting round matches the epoch declared in the vote.
- *BLS signature–verification delay.* Vote BLS signature verification is delayed until QC construction because verifying BLS signatures is expensive, and under block-time constraints, verifying each vote signature individually is not feasible.

State updates and subsequent operations. After receiving votes, the leader stores them in a pending votes mapping (`pending_votes`) and detects any double-voting behavior. When a proposal reaches a supermajority of votes, the leader attempts to aggregate signatures into a SignatureCollection. If signature aggregation succeeds, it creates the corresponding QC and updates `earliest_round` to avoid building QCs for the same round repeatedly. If invalid signatures exist, it removes these invalid signatures from pending votes and continues checking whether supermajority is still reached. QC creation marks consensus achievement for that round, and the new QC will be referenced in subsequent rounds' `Proposal`s, advancing the entire consensus process.

3. Message type: `Timeout`

This section outlines the structure, basic functions, and life cycle of `Timeout` messages.

Message structure and basic functions

The `Timeout` messages indicate validators abandoning the current round and requesting to enter the next round. The core function is to allow nodes to coordinate `this round has failed` when a round fails to form a `Proposal` or QC on time, thereby triggering view change (round progression). These `Timeout` messages carry time-out information (`TimeoutInfo`), high-extend information (`HighExtend`), and optional previous round TCs, providing necessary information for subsequent aggregation into certificates.

Moreover, `HighExtend` may contain QC or tip. If a node voted for a block in the current round, it carries that tip and voting signature; otherwise, it carries the highest QC. The key purpose of carrying the tip is to implement a block re-proposal mechanism: when `2f + 1` nodes' `Timeouts` converge into a TC, if the TC's high-extend information contains the same tip, the new round's leader can directly re-propose the block corresponding to that tip based on the TC, rather than proposing an entirely new block. This ensures that even if a round times out due to network delays, blocks that have received partial voting support still have a chance to complete consensus in subsequent rounds, avoiding discarding valuable proposals due to single time-outs.

Message life cycle

Creation and propagation. The triggering condition is local timer time-out; if validators wait for a leader's `Proposal` in the current round for more than the predetermined time without receiving a valid proposal or completing consensus, they consider that the round may have failed (such as a leader being offline or network being unreachable). At this time, the Pacemaker module drives this node to construct a `Timeout` message containing current epoch/round, highest QC or tip information, and optional previous round TC certificate and then broadcasts to all other validators, indicating "I have abandoned round r".

Reception and validation. When processing `Timeout` messages, nodes need to verify message legality, including the following validation steps:

- *Message-integrity verification.* Using EpochManager and `val_epoch_map` to verify the `Timeout` message's digital signature, ensure the message was issued by the sender.

- *Epoch verification.* Check that the message's epoch matches with the round's, ensuring the message comes from the correct epoch and the declared round is legal.
- *TC validity verification.* If the message contains `last_round_tc`, verify the TC's signature collection validity and ensure the TC round is exactly the current round minus 1, preventing malicious nodes from forging TC information.
- *High-extend information verification.* Verify QC or tip information contained in `high_extend`, ensuring referenced certificates are legally valid, and record this information for subsequent safety judgment.

State updates and subsequent operations. When receiving nodes process `Timeout` messages, they record the sender's time-out stance for round r and store it in `pending_timeouts`, while first processing QC information carried in the message, then checking whether `HighExtend` contains tip and voting signatures, treating it as a `Vote` message if present.

When receiving `f + 1` same-round `Timeouts`, if this node has not timed out yet, they trigger local time-out behavior. When receiving `2f + 1` `Timeouts`, they attempt to aggregate signatures to create a TC. After successful TC creation, they call `process_certificate` to advance to the next round — Pacemaker resets phase state and updates the current round. If invalid signatures exist, they remove corresponding invalid `Timeout` messages from `pending_timeouts` and continue checking whether supermajority is still achieved. TC serves as proof for entering a new round, ensuring all nodes entering round `r + 1` know that round r has failed, while providing important information for the new leader on how to safely start the new round, thus achieving view-change functionality, allowing the system to progress consensus even without forming a QC.

4. Message type: `RoundRecovery`

This section outlines the structure, basic functions, and life cycle of `RoundRecovery` messages.

Message structure and basic functions

The `RoundRecovery` messages are used for a tip-block recovery confirmation mechanism. When a new round's leader wants to re-propose the previous round's tip block based on a TC but discovers the tip block is unavailable (such as a locally missing, inconsistent, or invalid timestamp), they send `RoundRecovery` messages to query other nodes in the network about whether they have that tip block. The `RoundRecovery` messages carry the target round, corresponding epoch, and TC containing tip information. The core function is to confirm whether tip blocks can be recovered; if not recoverable, it collects `NoEndorsement` messages to form NEC and safely skip that tip.

Message life cycle

Creation and propagation. The triggering condition is when the new round's leader discovers during the `try_propose()` process that the tip block contained in the TC cannot be re-proposed — tip-block consistency check fails (`!is_coherent`) or timestamp validation fails (`!is_votable_ts`). At this time, the leader simultaneously takes two actions: 1) starting block-synchronization requests through `request_tip_if_missing()` to try obtaining the missing tip block, while 2) immediately sending `RoundRecovery` messages to query other nodes in the network about whether they have that tip block. The two processes run in parallel, with block synchronization for data recovery and `RoundRecovery` for confirming recovery possibility. The message contains the current round, epoch, and TC containing tip information, broadcasting to all validators.

Reception and validation. When validators receive `RoundRecovery` messages, they need to verify message legality, including the following validation steps:

- *Message-integrity verification.* Use EpochManager and `val_epoch_map` to verify the `RoundRecovery` message's digital signature, ensuring the message was issued by the sender.
- *Leader-identity verification.* Verify through LeaderElection that the sender is indeed the legitimate leader for the declared round, and the round must match the current Pacemaker round.
- *TC validity verification.* Verify whether the TC carried by the message is authentic and valid, checking that the TC's `high_extend` must contain the tip (cannot be QC), ensuring this is a scenario requiring tip recovery.
- *Epoch verification.* Check the message epoch's matching with rounds, ensuring the message comes from the correct epoch.

State updates and subsequent operations. After validators pass `RoundRecovery` message validation, they first advance local round state through the TC. Then, they check the tip block in the TC: if the local tip block is consistent and the timestamp is valid, they ignore the request (indicating the tip is available); if the tip block is inconsistent or the timestamp is invalid, and safe no-endorsement sending conditions are met, they send `NoEndorsement` messages to the requesting leader, indicating an inability to provide that tip block.

The safety condition requires that the current round must be higher than the maximum of this node's historical highest no-endorsement round and highest voting round, meaning the round is simultaneously higher than the highest round of previously sent no-endorsement messages and the highest round of previous votes, preventing repeated no-endorsement or conflicts with previous votes. After leaders collect enough `NoEndorsement` messages, they can form a no-endorsement certificate (NEC), proving the tip block cannot be recovered, thus safely skipping that tip and proposing new proposals based on QC information in the TC. This mechanism ensures that even when some tip blocks are lost due to network issues or leader failures, the protocol can still safely continue progressing, avoiding protocol stagnation due to single tip block loss.

5. Message type: `NoEndorsement`

This section outlines the structure, basic functions, and life cycle of `NoEndorsement` messages.

Message structure and basic functions

The `NoEndorsement` messages are validators' responses to `RoundRecovery` requests, indicating their inability to provide the requested tip block. The core function is to provide "I don't have this block" feedback when leaders inquire about tip-block availability. The `NoEndorsement` messages include epoch, round, tip-block ID, and the QC round corresponding to the tip. Multiple validators' `NoEndorsement` messages can be aggregated to form a NEC, providing basis for new leaders to safely skip unrecoverable tips.

Message life cycle

Creation and propagation. The triggering condition is when validators receive `RoundRecovery` requests and discover they cannot provide the requested tip block. The specific scenario is as follows: after validators receive a TC containing tip and `RoundRecovery` messages, they check local tip block status, and if the tip block is inconsistent or the timestamp is invalid, and safe no-endorsement sending conditions are met, they construct `NoEndorsement` messages for point-to-point sending to the requesting leader. Messages are not broadcast network wide but directly reply to the leader who sent the `RoundRecovery`.

Reception and validation. When leaders receive `NoEndorsement` messages, they need validation, including the following validation steps:

- *Message-integrity verification.* Use EpochManager and `val_epoch_map` to verify the NoEndorsement message's digital signature, ensuring the message was issued by the sender.
- *Epoch verification.* Check the epoch matching with rounds in the message, ensuring epoch information is correct.
- *Round check.* Verify the message round is not less than the current round; outdated `NoEndorsement` messages are discarded.

State updates and subsequent operations. Leaders collect and process `NoEndorsement` messages through the NoEndorsementState module. When receiving enough `NoEndorsement` messages with weight reaching supermajority, the system aggregates these signatures to form NECs. NEC formation indicates the tip block cannot be recovered and leaders can safely skip that tip. In subsequent proposal processes, if available NECs exist, leaders use the QC portion from TC as `high_extend` and use a NEC as FreshProposalCertificate to construct new proposals, rather than re-proposing the original tip block. This mechanism ensures that even when some tip blocks cannot be recovered due to network issues or failures, the protocol can still safely continue progressing, avoiding protocol stagnation due to single tip-block loss.

Consensus state machine

Monad Consensus builds a message-driven state machine based on five core message types, with its operation following this cycle: leader proposes → validators vote → aggregate to form QC → two QCs trigger commit → time-out forms TC → propose based on TC content (re-propose tip or propose new block based on QC). The protocol exhibits different execution paths under different scenarios:

Normal execution path. Each round begins with a `Proposal`, validators send `Vote`s to the next round's leader after validation, and the new leader aggregates votes to form a QC. Two consecutive QCs enable block commits and continuous chain growth. During this process, time-out mechanisms are not triggered, and the protocol operates efficiently.

Time-out recovery path. When a round cannot reach consensus, validators broadcast `Timeout` messages indicating abandonment of the current round. Also, `HighExtend` carries tip information supporting block re-proposal mechanisms. Receiving `f + 1` `Timeouts` triggers local time-out behavior, and receiving `2f + 1` `Timeouts` forms a TC and advances to the next round. New leaders decide subsequent operations based on TC content: if containing a tip, they attempt re-proposal or start the recovery process; if containing only a QC, they directly propose new blocks based on the QC.

Block recovery path. This is triggered only when a TC's `high_extend` contains a tip. When new leaders discover TC's tip block cannot be directly re-proposed, they simultaneously launch block synchronization and `RoundRecovery` query mechanisms. If majority nodes cannot provide the tip, they collect `NoEndorsement` to form a NEC proving the tip is unrecoverable, and safely skips and proposes new blocks; if tip-block consistency checks and timestamp validation pass, they directly re-propose that block to complete the consensus process.

This design ensures the protocol can safely progress under various network conditions, neither stagnating due to failures nor discarding blocks that have achieved partial consensus.

6. Assessment Results

During our assessment on the scoped Monad targets, we discovered 11 findings. Five critical issues were found. Two were of high impact, one was of medium impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.