Zellic

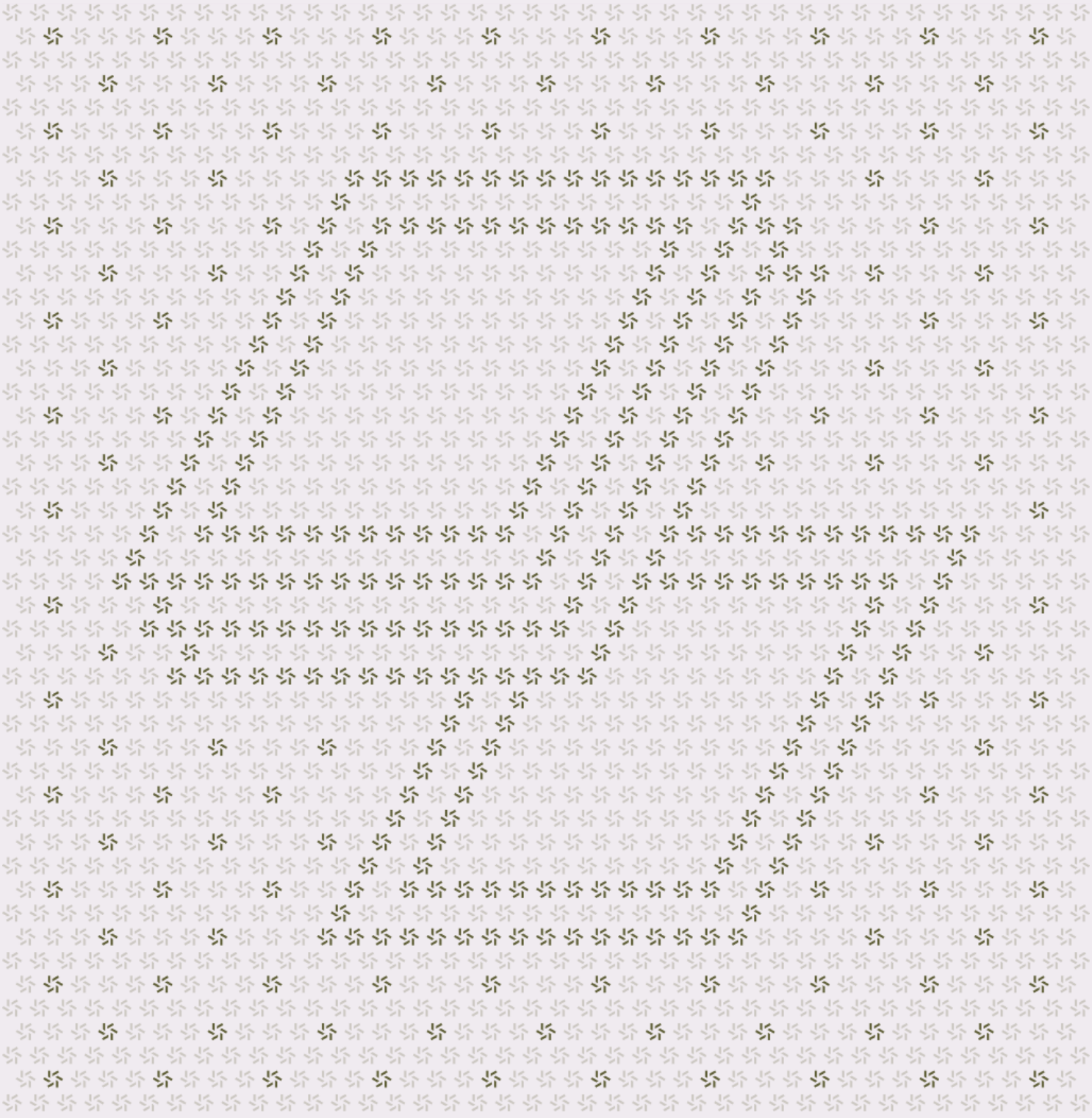
August 7, 2025

Prepared for
James Hunsaker
Category Labs, Inc.

Prepared by
Avi Weinstock
Ayaz Mammadov
Bryce Casaje
Can Boluk
Chongyu Lv
Filippo Cremonase
Jinseo Kim
Kuilin Li
Nan Wang
Ziling Chen
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. 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:

- Is there any remote denial-of-service (DOS) vulnerabilities that could crash or take down nodes/validators?
- Are there any ways for unauthenticated persons to impersonate/spoof name records that represent validators/nodes?
- Are there any issues that could halt peer discovery, resulting in centralization?
- Are there any issues that could result in the incorrect encoding/decoding of UDP messages, whether due to Merkle tree issues or incorrect raptor encoding/decoding?

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
- Node availability

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. Four critical issues were found. Two were of high impact, three were of medium impact, one was of low impact, and the remaining finding was 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	4
High	2
Medium	3
Low	1
Informational	1

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

Programs	`monad-dataplane/src/*.rs`
	`monad-peer-discovery/src/*.rs`
	`monad-peer-disc-swarm/src/*.rs`
	`monad-raptorcast/src/*.rs`
	`monad-raptor/src/*.rs`
	`monad-router-filter/src/*.rs`
	`monad-router-scheduler/src/*.rs`
	`monad-node/src/*.rs`
	`monad-eth-txpool/src/*.rs`
	`monad-eth-txpool-executor/src/*.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 46.6 person-weeks. The assessment was conducted by ten consultants over the course of nine 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:

Can Boluk
Engineer
can.boluk@zellic.io

Bryce Casaje
Engineer
bryce@zellic.io

Ziling Chen
Engineer
ziling@zellic.io

Filippo Cremonese
Engineer
fcremo@zellic.io

Jinseo Kim
Engineer
jinseo@zellic.io

Kuilin Li
Engineer
kuilin@zellic.io

Chongyu Lv
Engineer
chongyu@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

Nan Wang
Engineer
nan@zellic.io

Avi Weinstock
Engineer
avi@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
September 5, 2025	End of primary review period

3. Detailed Findings

3.1. Raptorcast remote node panic via malformed group message

Target	raptorcast-secondary		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When processing `PrepareGroup` messages in raptorcast_secondary/client.rs at `on_receive_group_message():146`,

```
// Sanity check the message
if invite_msg.start_round >= invite_msg.end_round {
    tracing::warn!(
        "RaptorCastSecondary rejecting invite message due to \
        failed sanity check: {:?}",
        invite_msg
    );
    accept = false;    // <-- Sets a flag but continues processing!
}
```

the code detects invalid bounds but continues iterating instead of returning early. This causes an immediate panic here:

```
// Check confirmed groups
for group in self
    .confirmed_groups
    .values(invite_msg.start_round..invite_msg.end_round)  // <-- panics when
    start > end

>> inside iset crate:
 216   | fn check_interval<T: PartialOrd + Copy>(start: T, end: T) {
 217   |       if start < end {
 218   |          assert!(end > start, "Interval cannot be ordered (`start <
    end` but not `end > start`)");
 219   |       } else if end <= start {
 220   |           panic!("Interval is empty (`start >= end`)");
 221   |       } else {
 222   |           panic!("Interval cannot be ordered (not `start < end` and not
    `end <= start`)");
 223   |       }
```

```
 224    |  }
2025-07-11T16:33:25  WARN monad_raptorcast::raptorcast_secondary::client:
    RaptorCastSecondary rejecting invite message due to failed sanity check:
    PrepareGroup { validator_id: ..., max_group_size: 10, start_round: 30,
    end_round: 20 }
```

Impact

Any malicious node can crash all full nodes in the network with a single malformed packet.

Recommendations

We recommend the following.

- Fail fast when user data is malformed.
- Do not use `assert!` in places user input can reach; it should be reserved for boundaries where it is clear developer violation of a library contract.
- Consider implementing `TryFrom<(Round, Round)>` for `RoundSpan` checking for invalid bounds. A very similar `assert!` can be found at `RoundSpan::new`.

Remediation

This was remediated in commit f50f6622 ↗ by returning false instead of continuing when the `PrepareGroup` date is malformed/invalid.

3.2. Arbitrary client-controlled compression-policy denial of service via resource exhaustion

Target	monad-raptorcast		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When processing TCP messages in lib.rs at the router level, the dataplane packet is passed directly to deserialization:

```
        while let Poll::Ready((from_addr, message)) =
            pin!(this.dataplane.lock().unwrap().tcp_read()).poll_unpin(cx)
        {
            // check message length to prevent panic during message slicing
            if message.len() < SIGNATURE_SIZE {
                warn!(
                    ?from_addr,
                    "invalid message, message length less than signature size"
                );
                continue;
            }
            let signature_bytes = &message[..SIGNATURE_SIZE];
            let signature = match ST::deserialize(signature_bytes) {
                Ok(signature) => signature,
                Err(err) => {
                    warn!(?err, ?from_addr, "invalid signature");
                    continue;
                }
            };
            let app_message_bytes = message.slice(SIGNATURE_SIZE..);
            let deserialized_message =
                match InboundRouterMessage::<M,
    ST>::try_deserialize(&app_message_bytes) { // <-- dataplane packet passed
    straight to router
Inside try_deserialize(), the sender controls the entire network policy:
impl<M: Decodable, ST: CertificateSignatureRecoverable>
    InboundRouterMessage<M, ST> {
    pub fn try_deserialize(data: &Bytes) -> Result<Self, DeserializeError> {
        let mut data_ref = data.as_ref();
        let mut payload =
```

```
            Header::decode_bytes(&mut data_ref,
    true).map_err(DeserializeError::from)?;
        if !data_ref.is_empty() {
            return Err(DeserializeError("extra data after header".into()));
        }
        let version =
            NetworkMessageVersion::decode(&mut
    payload).map_err(DeserializeError::from)?; // <-- network policy & limits
    decided by sender!!
        let message_type = u8::decode(&mut
    payload).map_err(DeserializeError::from)?;
        let result = match message_type {
The NetworkMessageVersion struct allows arbitrary limits:
struct NetworkMessageVersion {
    pub serialize_version: u32,
    pub compression_version: CompressionVersion,
    pub max_message: usize,  // <-- sender controls this!
}

const MAX_MESSAGE_SIZE: usize = u32::MAX as usize;

impl NetworkMessageVersion {
    pub fn version() -> Self {
        Self {
            serialize_version: SERIALIZE_VERSION,
            compression_version: CompressionVersion::UncompressedVersion, //
    <-- no users of compression (?), but feature exposed to mainnet clients
            max_message: MAX_MESSAGE_SIZE,  // <-- 4GB by default
        }
    }
}
```

The uncompressed version does not enforce any limits — as the dataplane is responsible for that — but naturally, when `CompressionVersion::DefaultZSTDVersion` is specified, the responsibility falls on the router, which uses the sender's limit for the memory allocations:

```
let decompressed_message_len =
    u32::decode(&mut payload).map_err(DeserializeError::from)?;
if decompressed_message_len as usize > version.max_message {  // <-- using
    sender's version.max_message!
    return Err(DeserializeError(format!(
        "message size {} exceeds maximum allowed size {}",
        decompressed_message_len, version.max_message
    )));
}
The code then pre-allocates the entire claimed size:
```

```
let mut decompressed_writer = BoundedWriter::new(decompressed_message_len);
    // <-- full allocation!
ZstdCompression::default()
    .decompress(&compressed_app_message, &mut decompressed_writer)  // <--
    Zstd writes every page
```

Impact

A malicious attacker could exploit this vulnerability by establishing multiple connections (up to 100) and sending crafted 64 KB packets that each claim to decompress to 4 GB, forcing immediate memory allocation. This attack simultaneously consumes resources through multiple vectors: massive memory allocations, page faults triggered during the decompression process, computational overhead from cryptographic hashing, and potential CPU core saturation if zstd worker threads are enabled. This would almost certainly result in a DOS either via memory exhaustion or via unresponsiveness due to the page faults combined with the decompression process, which takes up to ~1 second per packet sent.

Recommendations

We recommend the following.

- Enforce a server-defined `MAX_DECOMPRESSED_SIZE` regardless of the client's `version.max_message`.
- Validate the zstd header instead of / as well as limiting writes.
- Add per-connection memory quotas before accepting compressed messages.

Remediation

This was remediated in commit 0c0f63ff ↗ by changing the logic to use a constant server defined `MAX_MESSAGE_SIZE` instead of a user controlled data.

3.3. Message `PrepareGroup` bandwidth-check bypass

Target	raptorcast-secondary		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The message `PrepareGroup`, which prepares an invite for a group, checks whether or not bandwidth requirements are enforced.

```
pub fn on_receive_group_message(...) {
  let mut future_bandwidth:
    Bandwidth =             self.bandwidth_cost(invite_msg.max_group_size);

        fn bandwidth_cost(&self, group_size: usize) -> Bandwidth {
        group_size as u64 * self.config.bandwidth_cost_per_group_member
    }
```

These calculations are vulnerable to integer overflow from multiplication if `config.bandwidth_cost_per_group_member > 1`. Bandwidth can also overflow if there are pending invites/confirmed groups and calculating the total bandwidth causes an overflow from addition, as seen below.

```
// Check groups we were invited to but are still unconfirmed
              for (&key, other_invites) in self.pending_confirms.iter() {
                  if key >= invite_msg.end_round {
                      // Remaining keys are outside the invite range
                      break;
                  }
                  for other in other_invites.values() {
                      if Self::overlaps(other.start_round, other.end_round,
    &invite_msg) {
                          future_bandwidth +=
    self.bandwidth_cost(other.max_group_size)
                      }
                  }
              }
```

Impact

A malicious validator can cause a DOS to any node by making a very large raptorcast group, causing a very large amount of rebroadcasts, leading to resource exhaustion.

Recommendations

Fix the integer overflows, which would prevent the bandwidth bypass.

Remediation

This was remediated in commit f50f6622 ↗ by adding logic that detects overflows.

3.4. Memory exhaustion via preallocated message buffers

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

Description

> **Note:** The Category Labs, Inc. team was aware of this issue at the time of the audit, and we confirmed it independently. As it was present in the reviewed commit, we have documented it here for completeness.

In `tcp/rx.rs::read_message()`, incoming buffers are preallocated based on user-supplied length fields:

```
let len = header.length.get() as usize;
... // asserted to be < TCP_MESSAGE_LENGTH_LIMIT (1GB)

let buf = BytesMut::with_capacity(len); // reserved immediately
...
timeout(message_timeout(len), tcp_stream.read_exact(buf)).await;
```

The time-out calculation assumes 1 MB/sec minimum throughput, giving attackers 16m 40s for a 1 GB message. This enables multiple attack vectors.

This first is **resident memory exhaustion**. To trigger page faults, 1 GB minus 4 KB is sent, loading buffer into RAM. An attacker idles until time-out while keeping memory resident. Each connection can allocate up to 100 GB (connection limit x 1 GB). A botnet with ~500 Mbps bandwidth can cause an out-of-memory condition for the node before time-outs occur.

The second is **virtual memory and page-table exhaustion**. Even without sending data, each 1 GB allocation creates ~300K page table entries plus 512 PDEs. Page table entries alone consume ~2 MB kernel memory per connection — 100K connections would attempt to allocate 200 GB in kernel memory, and the system would crash from page table exhaustion, not application memory.

Impact

This may lead to a complete node crash via the OOM or kernel-memory exhaustion. It may also lead to a network-wide DOS with modest botnet resources.

There is also an amplification factor — attackers send headers, and the node allocates gigabytes.

Combined with the IPv6 connection limit bypass (Finding 3.5. ↗), the impact becomes catastrophic as it removes the 100-connection-per-IP limitation that normally constrains this attack. Without connection limits, a single attacker can establish thousands of connections from a single IPv6 address, multiplying the memory exhaustion potential by orders of magnitude and making the attack feasible even without a botnet.

Recommendations

We recommend the following.

- **Per-client memory limits** — track total allocation per IP/prefix, not per connection.
- **Streaming consumption** — pass readers to L7 for on-demand processing instead of buffering entirely.
- **Reduce maximum message size** — require application-layer fragmentation for large payloads.
- **Progressive buffer allocation** — start small, and grow as data arrives with backpressure.
- **Accounting for kernel memory** — consider page-table overhead in resource limits.

Remediation

This was remediated in commit 0c0f63ff ↗ by limiting the max message size to `3MB`, this reduces the amount of memory pressure on the node even if targeted by a botnet.

3.5. Per-peer connection limit bypass via IPv6 address generation

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

Description

The connection handler enforces per-peer connection limits using the exact IP address as the bucket key:

```
fn check_new_connection(&self, ip: IpAddr) -> Result<(), ()> {
    let mut inner = self.inner.borrow_mut();
    let count = inner.num_connections.entry(ip).or_insert(0);

    if *count < PER_PEER_CONNECTION_LIMIT {
        *count += 1; // limit applies to *exact* IP
        Ok(())
    } else {
        Err(())
    }
}
```

This implementation has two critical flaws:

1. **IPv6 address space exploitation.** Consumer-grade ISPs typically lease /64 prefixes to clients, providing 2^{64} unique addresses. An attacker can generate different IPs from their prefix to create unlimited connections, each counted separately in the hashmap.
2. **IPv4-mapped IPv6 double counting.** IPv4 addresses can be represented as IPv6 using the `::ffff:0:0/96` prefix (e.g., `::ffff:192.168.1.1`). If the application accepts both protocols, a single IPv4 address effectively gets double the connection allowance.

Impact

Connection limits become meaningless against IPv6-capable attackers, and resource-exhaustion attacks bypass per-peer restrictions.

This issue may lead to memory DOS through unbounded connection creation.

Additionally, IPv4 attackers can double their connection allowance via mapping.

Recommendations

We recommend the following:

1. Normalize IPv4-mapped addresses, and convert `::ffff:a.b.c.d` back to IPv4 `a.b.c.d` before applying limits.
2. Use prefix-based buckets for IPv6, and group connections by /64 or /56 prefix rather than exact IP.
3. Consider implementing progressive rate limiting based on connection patterns.
4. Add monitoring for connection distribution across IP ranges.

Remediation

The Monad team states that IPv6 is not supported, and that `monad-dataplane` will be configured to only use IPv4.

3.6. Consensus stall via forced LRU cache eviction

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

Description

> **Note:** The Category Labs, Inc. team was aware of this issue at the time of the audit, and we confirmed it independently. As it was present in the reviewed commit, we have documented it here for completeness.

The broadcast handler's `pending_message_cache` uses a simple LRU eviction policy when the cache reaches capacity:

```
while self.pending_message_cache.len() > PENDING_MESSAGE_CACHE_SIZE.into() {
    let (key, decoder_state)
    = self.pending_message_cache.pop_lru().expect("nonempty");
    // Simply drops the oldest entry regardless of importance
}
```

This design allows low-stake validators to force eviction of high-stake validators' messages. This vulnerability exists due to the following factors:

1. **No stake-weighted protection.** The cache treats all messages equally, regardless of the sender's stake weight.
2. **Forced eviction attack.** A minimal-stake validator can flood the cache with junk messages.
3. **In-progress reconstruction loss.** Legitimate message reconstructions get evicted midprocess.
4. **Consensus impact.** Loss of high-stake validator messages can stall consensus.

The issue persists even with planned mitigations for memory bounds and timestamp validation, as the fundamental LRU policy remains exploitable.

Here is the attack scenario.

1. High-stake validators broadcast important consensus messages.

2. A malicious low-stake validator floods the network with valid but useless messages.
3. The LRU cache fills up and evicts the oldest entries.
4. Partially reconstructed messages from high-stake validators are dropped.
5. The network loses critical consensus messages, potentially stalling progress.

Impact

This may lead to consensus stalls from loss of high-stake validator messages, data-layer availability degradation, and network instability during critical consensus phases.

This type of attack is amplified — minimal stake causes maximum disruption.

Recommendations

We recommend the following.

- **Stake-weighted cache management.** Prioritize retention based on sender's stake.
- **Per-validator limits.** Cap cache entries per public key.
- **Adaptive eviction policy.** Consider multiple factors: the sender's stake weight, message reconstruction progress (percent complete), message type priority (consensus vs data), and network conditions (high vs low activity).
- **Reserved capacity.** Guarantee minimum slots for high-stake validators.

Remediation

TBD

3.7. Lack of domain separation in Merkle proofs

Target	monad-raptorcast		
Category	Business Logic	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

The function `parseMessage` is responsible for receiving fragments of messages due to the natural limits of UDP and message size. A Merkle tree is used to deem which fragments are valid.

```
pub fn parse_message<ST>(
    signature_cache: &mut LruCache<
        [u8; HEADER_LEN as usize + 20],
        NodeId<CertificateSignaturePubKey<ST>>,
    >,
    message: Bytes,
) -> Result<ValidatedMessage<CertificateSignaturePubKey<ST>>,
    MessageValidationError>
where
    ST: CertificateSignatureRecoverable,
{
    ...

    let leaf_hash = {
        let mut hasher = HasherType::new();
        hasher.update(
            &message[HEADER_LEN as usize + proof_size as usize..
                // HEADER_LEN as usize
                //     + proof_size as usize
                //     + CHUNK_HEADER_LEN as usize
                //     + payload_len as usize
                ],
        );
        hasher.hash()
    };
    let root = merkle_proof
        .compute_root(&leaf_hash)
        .ok_or(MessageValidationError::InvalidMerkleProof)?;

}
```

However, there is no logic that separates leaves from nodes in the Merkle tree; as such, a proof can be generated for a Merkle tree without the original leaf data.

Impact

In the case of this specific Merkle proof construction, another property of the messages, `merkle_tree_depth`, prevents a shorter or longer proof from being supplied. This means that no alternative Merkle proof can be forged for the root.

Recommendations

Integrate domain separation between the leaves and the nodes of the Merkle tree when calculating the root.

Remediation

TBD

3.8. Blind server-side request forgery through unvalidated `NameRecord` / `PeerEntry` records

Target	monad-peer-discovery		
Category	Business Logic	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

Any node in the network can advertise arbitrary IPv4 addresses in their `NameRecord` through multiple vectors, including `Ping` messages and `ConfirmGroup` messages, which are then blindly accepted and used as destinations for network traffic. This allows attackers to use validator nodes as one-way packet cannons to target internal infrastructure, cloud metadata services, or other restricted network resources.

See below:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct NameRecord {
    pub address: SocketAddrV4,  // <- No validation during decode time
    pub seq: u64,
}
```

This decoding happens in three separate places:

1. Handling a `Ping` from a new node
2. Group logic, when confirming a group received from a validator
3. Via the `UpdatePeers` mechanism exposed in peer discovery

Impact

A malicious attacker could supply `NameRecords` / `PeerEntrys` that point to internal IPs accessible only to machines in the internal network. Any messages that are rebroadcast or emitted responses such as `pong` could reach `DNS (53)`, `DHCP (67/68)`, `LDAP (389)`, `mDNS (5353)`, `NetBIOS (137-139)`, `UPnP/SSDP (1900)`, `Syslog (514)`, `SNMP (161)` and other services on private networks. While the contents of the messages that are sent to the internal services are not controllable, certain services may react abnormally to unexpected protocol messages.

This constitutes a blind server-side request forgery (SSRF) vulnerability, where the validator nodes act as unwitting proxies to access internal network resources that would otherwise be

unreachable from external networks.

Recommendations

Types like `PeerEntry` and `NameRecord` should be validated when received across the network boundary given the current configuration (i.e., a local address might be appropriate in an end-to-end testing environment, but it should be rejected in production deployments).

Remediation

TBD

3.9. Missing replay protection in message validation

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

Description

> **Note:** The Category Labs, Inc. team was aware of this issue at the time of the audit, and we confirmed it independently. As it was present in the reviewed commit, we have documented it here for completeness.

The `parse_message` function validates signatures and message structure but lacks replay-attack protection:

```
pub fn parse_message<ST>(
    signature_cache: &mut LruCache<
        [u8; HEADER_LEN as usize + 20],
        NodeId<CertificateSignaturePubKey<ST>>,
    >,
    message: Bytes,
) -> Result<ValidatedMessage<CertificateSignaturePubKey<ST>>,
    MessageValidationError>
where
    ST: CertificateSignatureRecoverable
```

The function validates the signature to authenticate the sender, parses the Unix timestamp from the message, and does **not** check if the timestamp is fresh or if the message was seen before.

The timestamp is only used as part of the hash computation and never validated for freshness:

```
let cursor_unix_ts_ms = split_off(8)?;
let unix_ts_ms = u64::from_le_bytes(
    cursor_unix_ts_ms
        .as_ref()
        .try_into()
        .expect("u64 is 8 bytes"),
);
// unix_ts_ms extracted but never checked for validity/freshness
```

This allows attackers to 1) capture legitimate signed messages from validators and 2) replay them indefinitely to flood the network.

Impact

This may lead to network flooding through unlimited message replay as well as resource exhaustion from processing duplicate messages.

Recommendations

Add timestamp validation, and reject messages older than a configurable threshold.

Remediation

This issue was remediated in commit abff695a ↗ by adding a function called `ensure_valid_timestamp` invoked in the `parse_message` functionality. This enforces the message to be sent within a time delta of its timestamp consequently adding replay protection to old messages.

3.10. Raptor parameters do not accomodate small values

Target	monad-raptor/src/r10/parameters.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

In raptor, the values `X` and `H`, computed using the methods `determine_x` and `determine_num_half_symbols`, are used to calculate the number of LDPC and half symbols needed by raptor code encoding a fixed number of source symbols.

According to the raptor RFC, the minimum values for `X` and `H` are `4` and `5`, respectively. These correspond to the lowest number of source symbols for raptor code, which is `K = 4`. Monad's raptor implementation deviates from the specification and allows less than four source symbols to be encoded. However, the `X_MIN` and `HALF_MIN` values are left unchanged.

```
const X_MIN: usize = 4;
const X_MAX: usize = 129;

fn determine_x(num_source_symbols: u16) -> Result<u8, String> {
    match smallest_integer_satisfying(Self::X_MIN, Self::X_MAX + 1, |pivot| {
        pivot * (pivot - 1) >= (2 * num_source_symbols).into()
    }) {
        None => Err(format!(
            "Can't find x for num_source_symbols = {}",
            num_source_symbols
        )),
        Some(x) => Ok(u8::try_from(x).unwrap()),
    }
}
```

```
const HALF_MIN: usize = 5;
const HALF_MAX: usize = 16;

fn determine_num_half_symbols(
    num_source_symbols: u16,
    num_ldpc_symbols: u16,
) -> Result<u8, String> {
    let num_half_symbols =
        smallest_integer_satisfying(Self::HALF_MIN, Self::HALF_MAX + 1,
```

```
    |pivot| {
            let pivot = pivot.try_into().unwrap();

            Self::choose(pivot, (pivot + 1) / 2) >= num_source_symbols +
    num_ldpc_symbols
        });

    match num_half_symbols {
        None => Err(format!(
            "Can't find num_half_symbols for num_source_symbols = {},
    num_ldpc_symbols = {}",
            num_source_symbols, num_ldpc_symbols
        )),
        Some(num_half_symbols) => Ok(num_half_symbols.try_into().unwrap()),
    }
}
```

Impact

Since the values for `X_MIN` and `HALF_MIN` are not adjusted for `num_source_symbols` being less than four, it means that the encoding plaintext in that range will get assigned a higher value of correction symbols than is otherwise required. Additionally, the correction symbols are expected to contain duplicate values. For example, the `g_ldpc` method generates a triple of LDPC symbols to be used for each source symbol.

```
pub fn g_ldpc(&self, mut set_element: impl FnMut(usize, usize)) {
    for i in 0..self.num_source_symbols() {
        let mut b: [usize; 3] = self.ldpc_triple(i).into();

        b.sort();

        for el in b {
            set_element(el, i);
        }
    }
}
```

In the case of `self.num_source_symbols() = 1`, all three LDPC symbols will contain the same value, which is the same as the source symbol itself.

Recommendations

We recommend adjusting the value of `X_MIN` and `HALF_MIN` to smaller values to accommodate `K = 1,2,3`.

Alternatively, simply documenting the behavior is also sufficient as the finding does not currently have any security impact and only leads to minor performance boost.

Remediation

This issue has been acknowledged by Category Labs, Inc.

3.11. Raptor parameters do not accommodate small values

Target	monad-raptor/src/matrix/invert.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The method `DenseMatrix::check_invertible` is used to determine if a given matrix is invertible. The function is implemented as follows and simply checks that the result of running gaussian elimination on the matrix is successful.

```
pub fn check_invertible(self) -> bool {
    self.rowwise_elimination_gaussian(|_| {}).is_ok()
}
```

However, the function does not check if the matrix is square, and `rowwise_elimination_gaussian(|_| {}).is_ok()` may return true in the case where the number of rows in a matrix is larger than the number of columns.

Impact

The method `DenseMatrix::check_invertible` may return true for noninvertible matrixes when `nrows` is greater than `ncols` and a submatrix with `ncols` rows and columns is invertible. This, however, does not lead to any impact in the current codebase (as of the time of writing) as this function is not used anywhere outside of tests.

We recommend fixing this for potential future use cases.

Recommendations

We recommend changing the function as follows.

```
pub fn check_invertible(self) -> bool {
    self.nrows == self.ncols
    && self.rowwise_elimination_gaussian(|_| {}).is_ok()
}
```

Remediation

This issue has been acknowledged by Category Labs, Inc., and fixes were implemented in the following commits:

- aa18ecd2 ↗
- 36d49922 ↗

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. Peer discovery centralization

The peer discovery implementation focuses on validator discovery, with "open discovery" requests returning validator name records exclusively. Full nodes are not directly discoverable through the protocol, creating a hierarchical topology where validators serve as primary connection points for the network. This design means full nodes depend on validators for both consensus data and network topology information.

```
// if open discovery, return more nodes other than requested nodes
if request.open_discovery {
    let is_validator = |node_id:
    &NodeId<CertificateSignaturePubKey<ST>>| {...};

    name_records.extend(
        self.peer_info
            .iter()
            .filter(|(node_id, _)| is_validator(node_id))  // <-- only
    validators
            .choose_multiple(
                &mut self.rng,
                MAX_PEER_IN_RESPONSE.saturating_sub(name_records.len()),
            )
            .into_iter()
            .map(|(_, peer)| peer.name_record),
    );
}
```

The refresh mechanism targets validator discovery based on network design assumptions:

```
if self.peer_info.len() < self.min_active_connections {
    for (validator_id, peer)
    in missing_validators.iter().zip(chosen_peers.iter()) {
        cmds.extend(self.send_peer_lookup_request(*peer, *validator_id,
    true));  // <-- target always validator
    }
}
```

This design prevents horizontal peer discovery among full nodes. When a node restarts with an outdated validator set, it must rely on validators or manually configured bootstrap nodes to recover current topology.

Full nodes do learn about each other when participating in secondary raptorcast groups through their upstream validator coordination, providing an additional discovery path within these groups.

However, since peer records are not automatically persisted across restarts, nodes lose all discovered topology information and must bootstrap through manually configured peers. This creates a bottleneck at startup where nodes depend on bootstrap configuration remaining current and available.

We recommend implementing automatic periodic persistence of peer records and relaxing the rules for pruning validators to improve network resilience while maintaining the validator-focused architecture for consensus operations.

4.2. No liveliness checks on peer discovery

There are systems in place to detect if nodes are responsive. One of these systems ensures this invariant by enforcing that pings are responded to. Nodes that fail to respond to pings are pruned as they are no longer lively. However, this same system does not apply to nodes that are not responsive in peer discovery or raptorcast broadcasting.

Below is the ping time-out, responsible for incrementing the `unresponsive_pings` field of a peer, though the actual pruning happens later on.

```
fn handle_ping_timeout(
    &mut self,
    to: NodeId<CertificateSignaturePubKey<ST>>,
    ping_id: u32,
) -> Vec<PeerDiscoveryCommand<ST>> {
    ...
    // record timeout
    if let Some(last_ping) = peer_entry.last_ping {
        if last_ping.id == ping_id {
            debug!(?to, ?ping_id, "handling ping timeout");
            self.metrics[GAUGE_PEER_DISC_PING_TIMEOUT] += 1;
            peer_entry.unresponsive_pings += 1;
            peer_entry.last_ping = None;
        }
    }
    ...
}
```

Below is the peer-discovery request time-out. In this case, indeed there is the field `num_retries` that is incremented, but it is not used elsewhere in the codebase. It is only responsible for cancelling a peer-lookup request.

```
fn handle_peer_lookup_timeout(
    &mut self,
    to: NodeId<CertificateSignaturePubKey<ST>>,
    target: NodeId<CertificateSignaturePubKey<ST>>,
    lookup_id: u32,
) -> Vec<PeerDiscoveryCommand<ST>> {
    ...
    if lookup_info.num_retries >= self.prune_threshold {
        debug!(
            ?lookup_id,
            ?to,
            ?target,
            "peer lookup request exceeded number of retries, dropping..."
        );
        self.outstanding_lookup_requests.remove(&lookup_id);
        return cmds;

    }
    ...
```

As such, there might be a node that occupies the list of nodes but does not aid in gossip and broadcasting. The systems in place that detect responsive nodes should also account for this scenario by implementing additional pruning criteria:

Recommended solution: Implement comprehensive network participation tracking by extending the `PeerInfo` struct to monitor overall node responsiveness across all network activities. Specifically:

- Add `network_participation_failures` counter to track cumulative failures across all network operations (peer discovery timeouts, raptorcast broadcast failures, message forwarding errors, etc.)
- Increment this counter whenever a node causes timeouts or errors in any network protocol interaction
- Reset or decrement the counter when nodes successfully participate in network activities (successful lookups, chunk forwarding, message acknowledgments)
- Modify the pruning logic in `refresh()` to consider overall network participation

This generalized approach captures nodes that may respond to pings but fail to meaningfully participate in the broader network, whether through peer discovery unresponsiveness, raptorcast broadcasting failures, or other protocol-level issues.

4.3. Ping ID collision

There is no verification that the ID generated for the ping is unique (`send_ping`) such that no two consecutive pings can have the same `ping_id`. Consequently, a pong can respond to an old ping and still be valid. This is unlike `send_peer_lookup_request`, which ensures that a `unique_id` is generated when checking the RNG-generated ID.

```
fn send_peer_lookup_request(
    &mut self,
    to: NodeId<CertificateSignaturePubKey<ST>>,
    target: NodeId<CertificateSignaturePubKey<ST>>,
    open_discovery: bool,
) -> Vec<PeerDiscoveryCommand<ST>> {
    let mut cmds = Vec::new();

    // new lookup request
    let mut lookup_id = self.rng.next_u32();
    // make sure lookup id is unique
    while self.outstanding_lookup_requests.contains_key(&lookup_id) {
        lookup_id = self.rng.next_u32();
    }
```

Though this edge case seems unlikely, it would have an impact if a node sends a ping, changes its name record, and sends a ping with a new name record but accepts a pong for the old name record. In this scenario, the node would reset its `unresponsive_pings` counter based on a stale pong response, potentially keeping an outdated name record active and preventing proper timeout detection. This could lead to nodes maintaining connections to peers with outdated address information, disrupting network topology.

We recommend adding verification that ping IDs are unique by implementing the same collision-avoidance pattern used in `send_peer_lookup_request()`.

4.4. Style comments for Monad raptor

In this section we discuss possible improvements in style for Monad raptor.

binary_search.rs

The expression `lower + upper` could potentially overflow. Although the overflow will not be triggered by the current usage, we recommend adding a comment such as `Requires to <= usize::MAX/2` to prevent overflow in future use cases. Additionally, it should be made clear in the comment that the condition function should be monotone (i.e., if condition `x` is true, then condition `y` is true for all `y<=x`. This is crucial to the binary search being correct.

ordered_set.rs

It is the caller's responsibility to ensure that `append` is only called on an element that is greater than all elements in the set. We recommend removing the append function and replacing the usages with `insert`. This is slightly less efficient but much less error-prone.

r10/degree.rs

In this, `MAX_V` should be `1048575` instead of `1048576`. It is also not used anywhere, so we recommend directly removing this constant.

matrix/dense_matrix.rs

In `from_element` and `from_fn` of matrix/dense_matrix.rs, the expression `nrows * ncols` could potentially overflow. Although the overflow will not be triggered by the current usage, we recommend adding a comment such as `Requires nrows <= usize::MAX / ncols` to prevent overflow in future use cases.

r10/nonsystematic/buffer.rs

The comment before `xor_eq` stating caller is responsible for the bookkeeping is misleading.

- Based on the implementation of `xor_eq`, it might insert new symbols, increasing `self.active_used_weight` rather than decreasing it as the comment suggests.
- Multiple symbols could be removed, so `self.active_used_weight` may decrease by more than one. This is not an issue at the moment (as of the time of writing) because `xor_eq` is only used in receive_symbol.rs, which always calls `xor_eq` with `other` containing exactly one used symbol, so the `self.active_used_weight` will always decrease by one.

However, for clarity and future maintainability, we recommend revising the comment and using the following self-contained implementation. (To adopt this change, changes are also required to places where `xor_eq` is called, since this implementation now maintains `active_used_weight` directly.)

```
// Caller is responsible for dropping `self.active_used_weight` by 1 and
    performing
// any other necessary bookkeeping associated with that.
pub fn xor_eq(&mut self, other: &Buffer) {
        for intermediate_symbol_id in &other.intermediate_symbol_ids {
            if self.intermediate_symbol_ids
                .insert_or_remove(*intermediate_symbol_id) {
                self.active_used_weight += 1;
            } else {
                self.active_used_weight -= 1;
            }
                }
            }
        }
```

r10/nonsystematic/intermediate_symbol.rs

Several adjustments are recommended.

- The method `active_push` should take `buffer_index` as an `u16` instead of `usize`.
- Naming of the function `is_used_buffer_index` is inconsistent; we recommend changing it to `used_buffer_index`.
- The comment in `active_inactivated` about `BTreeSet` should be updated.

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. Networking design

This section documents the architecture and design choices of the Monad networking subsystem. The networking layer is composed of multiple specialized components that work together to provide message propagation across the validator and full node network.

Components

The Monad networking stack consists of the following major components.

monad-dataplane

This component handles OS-level network operations and protocol management. This includes socket configuration, connection management, message serialization/deserialization, and protocol-specific handling for TCP and UDP communications. It acts as the foundation layer for all network I/O operations.

monad-peer-discovery

This component implements a validator-centric gossip protocol for maintaining node connectivity and distributing signed name records. It provides ping/pong liveness detection, peer lookup requests with open/targeted discovery modes, and automatic pruning of unresponsive nodes. It maintains a hierarchical topology where full nodes depend on validators for network participation.

monad-raptorcast (primary)

This component is the core broadcasting protocol responsible for efficient message dissemination using forward error correction (FEC). It handles the following:

- Message fragmentation and reassembly using raptor FEC encoding
- Merkle tree–based authentication of message chunks
- Stake-weighted distribution to validators
- UDP-based chunk transmission with redundancy
- Compression and serialization of application messages

monad-raptorcast-secondary

This component manages secondary distribution groups where validators act as publishers and full nodes participate as group members. It reduces bandwidth load on validators by enabling peer-to-peer chunk sharing among full nodes within groups. It implements temporal validation and bandwidth-aware group formation.

monad-router-filter

This component provides message-filtering capabilities for full nodes to discard irrelevant consensus messages while allowing necessary synchronization traffic. It enables full nodes to participate in the network without processing validator-specific BFT consensus messages.

monad-raptor

This component is the implementation of the raptor FEC scheme (RFC 5053). It provides the underlying erasure coding functionality used by raptorcast for creating redundant message chunks that allow reconstruction even with packet loss.

Architecture principles

The networking design follows several key principles.

1. **Hierarchical topology.** Validators form the core mesh while full nodes connect as leaf nodes.
2. **Bandwidth efficiency.** FEC and group-based distribution minimize redundant transmissions.
3. **Message authentication.** Cryptographic signatures and Merkle proofs ensure message integrity.
4. **Selective participation.** Full nodes filter out consensus messages they do not need.
5. **Fault tolerance.** Redundancy and error correction handle packet loss and node failures.

Message flow

The typical message flow patterns in the Monad network follow these primary paths:

1. **Consensus messages.** These flow between validators using primary raptorcast with stake-weighted distribution.
2. **Block distribution.** This propagates from validators to full nodes via primary and secondary raptorcast.

3. **Transaction forwarding.** This allows full nodes to submit transactions through connected validators.
4. **Synchronization.** The `BlockSync` and `StateSync` requests flow bidirectionally for catch-up.

5.2. Component: monad-dataplane

Description

The Monad dataplane crate serves as the foundational networking layer for the Monad blockchain system, providing high-performance, asynchronous TCP and UDP communication capabilities. It acts as the network I/O foundation that other components build upon for peer-to-peer communication, block propagation, and consensus coordination.

Here are its core responsibilities.

- **Network protocol handling** — manages both TCP and UDP protocols with custom message framing
- **Connection management** — maintains persistent TCP connections with per-peer connection pooling and automatic reconnection
- **Resource management** — implements bandwidth-based pacing for UDP transmission and enforces various limits (connection counts, message-queue sizes, message sizes)
- **Asynchronous I/O** — utilizes `io_uring` through the Monoio runtime for efficient network operations in a dedicated thread
- **Message distribution** — provides channel-based communication between the high-performance I/O thread and application threads using `MPSC` channels

The dataplane spawns a dedicated thread running a Monoio (`io_uring`) runtime. It maintains four primary channels for bidirectional communication: TCP ingress/egress and UDP ingress/egress.

The TCP module implements connection limits, message size limits, and dynamic time-outs to prevent slowloris attacks. The UDP module provides bandwidth pacing, provides generic segmentation offload (GSO) support, and handles both broadcast and unicast patterns with configurable segment sizes.

These are its key features.

- Zero-copy message passing using `bytes::Bytes`
- Per-peer message queuing with backpressure handling
- Connection state management with automatic cleanup
- IPv4/IPv6 dual-stack support with address family mismatch handling
- Configurable socket buffer sizes for high-throughput scenarios

Invariants

- **Connection limits.** No more than a certain configured number of TCP connections per IP address shall be accepted simultaneously.
- **Message size bounds.** TCP messages shall not exceed a certain configured size.
- **Time-out enforcement.** Message time-outs shall be dynamically calculated based on message size with minimum time-out.
- **Memory allocation.** TCP message buffers are preallocated based on the header-length field.
- **Thread isolation.** Network I/O operations execute exclusively in the dedicated Monoio runtime thread.
- **Connection state.** Per-peer TCP connection counts are accurately tracked and decremented on connection termination.
- **Bandwidth pacing.** UDP transmission respects configured bandwidth limits through calculated delays.

Test coverage

Cases covered

- UDP communication — basic broadcast and unicast message transmission and reception
- TCP communication — rapid message transmission and slow transmission with completion tracking
- Connection recovery — TCP transmit task recovery after peer disconnection/time-out
- Queue management — TCP message-queue–limit enforcement and overflow handling
- Connection failures — TCP connection-failure handling and completion notification
- Stride variations — UDP transmission with varying segment sizes
- Address family handling — IPv4/IPv6 address family mismatch scenarios
- Buffer management — UDP buffer-size configuration and overflow behavior

Cases not covered

- Memory exhaustion — no tests for the critical TCP memory-allocation vulnerability
- IPv6 edge cases — no tests for IPv4-mapped IPv6 addresses potentially bypassing connection limits
- GSO failure handling — no tests for GSO fallback mechanisms
- Time-out edge cases — no tests for zero–time-out calculations or time-out bypass attempts

5.3. Component: monad-peer-discovery

Description

The Monad peer-discovery component implements a validator-centric gossip protocol responsible for maintaining network connectivity and distributing cryptographically signed name records throughout the network. The system operates on a hierarchical topology where validators form the backbone of the network, while full nodes depend on validators for network participation and discovery.

The implementation consists of three main architectural layers:

1. **Discovery engine** (`PeerDiscovery`) — core state machine that manages peer relationships, maintains routing tables, and executes discovery algorithms
2. **Driver** (`PeerDiscoveryDriver`) — async runtime wrapper that handles timer management, event processing, and command execution
3. **Message protocol** — RLP-encoded message types for ping/pong heartbeats and peer lookup operations

The peer-discovery component encompasses several key operational areas that work together to maintain network connectivity and facilitate node communication.

Name-record system. Each node maintains a `MonadNameRecord` containing an IPv4 socket address and monotonic sequence number, cryptographically signed to prevent spoofing. The sequence number ensures only newer records propagate through the network, preventing replay attacks.

Discovery mechanisms. These include the ping/pong protocol and peer lookup protocol.

- *Ping/pong protocol* — provides liveness detection, name-record distribution, and sequence-number validation with configurable intervals
- *Peer lookup protocol* — supports both targeted discovery (specific node requests) and open discovery (returns random validator samples)

Validator-centric topology. Validators actively discover all other validators and are never pruned regardless of responsiveness. Full nodes connect through either other dedicated full nodes or validators.

Refresh and pruning. Periodic maintenance performs three operations: it prunes unresponsive noncritical nodes based on a configurable threshold, manages connection watermarks (min/max active connections), and discovers missing validators from the current and next epoch sets.

Invariants

- **Cryptographic integrity.** All name records must be cryptographically signed and signatures must be verified before acceptance.

- **Sequence-number monotonicity.** Name-record sequence numbers must be strictly increasing — older sequence numbers are rejected.
- **Validator persistence.** Validators from current and next epochs are never pruned regardless of responsiveness.
- **Connection limits.** Peer lookup responses are bounded.
- **Signature recovery consistency.** The recovered public key from a name-record signature must match the claimed node identity.
- **Epoch consistency.** Only validators from current/next epoch sets are returned in open discovery responses.
- **Outstanding request tracking.** Each peer lookup request has a unique ID and time-out tracking to prevent stale responses.

Test coverage

Cases covered

- Basic ping/pong flow — tests successful ping transmission, pong reception, and state updates (`test_check_peer_connection`)
- Ping time-out handling — verifies unresponsive ping counter increments and eventual pruning
- Invalid pong rejection — tests dropping pong messages with incorrect ping IDs (`test_drop_pong_with_incorrect_ping_id`)
- Peer lookup operations — complete lookup request/response cycle including retry mechanisms (`test_peer_lookup`)
- Open discovery — tests returning random validators when target is not found (`test_peer_lookup_target_not_found`)
- Name-record updates — validates sequence-number–based record updates and rejection of stale records (`test_update_name_record_sequence_number`)
- Response validation — tests dropping invalid lookup responses not in outstanding requests (`test_drop_invalid_lookup_response`)
- Amplification protection — verifies rejection of responses exceeding maximum peer limit (`test_drop_lookup_response_that_exceeds_max_peers`)
- Pruning logic — tests removal of unresponsive peers while protecting validators and dedicated nodes (`test_prune`)
- Connection management — tests behavior below minimum (`test_below_min_active_connections`) and above maximum (`test_above_max_active_connections`) connection thresholds
- Record validation — comprehensive testing of ping record handling with various sequence-number scenarios (`test_ping_record`)

Cases not covered

- Byzantine behavior — no tests for handling malformed or adversarial messages beyond basic validation

- Network partition recovery — missing tests for behavior during and after network splits
- Epoch transition edge cases — limited testing of validator-set changes during epoch transitions
- Timer edge cases — missing tests for timer expiration during state transitions
- Signature-verification failures — limited testing of cryptographic validation edge cases
- Large network scenarios — no tests with realistic network sizes (hundreds/thousands of peers)

5.4. Component: monad-router-filter

Description

The monad-router-filter is a filtering wrapper that sits between the router and full nodes to selectively block or allow specific messages based on node type. It implements the `Executor` and `Stream` traits, acting as a transparent proxy that filters both incoming commands (`RouterCommand`) and dispatched events (`MonadEvent`).

For commands (via the `Executor` trait), the filter allows the following:

- `BlockSync` requests and responses (enables full nodes to sync with network state)
- `ForwardedTx` messages (allows transaction propagation during BFT consensus)
- `StateSync` messages (facilitates state synchronization)
- Validator-set management commands (`AddEpochValidatorSet`, `UpdateCurrentRound`)
- Peer directory operations (`GetPeers`, `UpdatePeers`, `GetFullNodes`, `UpdateFullNodes`)

The filter blocks direct consensus message publishing (`VerifiedMonadMessage::Consensus`) and `PublishToFullNodes` commands.

For events (via the `Stream` trait), the filter allows only consensus proposal messages (it blocks votes, time-outs, and other BFT events). All nonconsensus events pass through unchanged, and `MempoolEvent` messages are dropped entirely.

This design ensures full nodes receive necessary synchronization data without receiving consensus-related communications.

Test coverage

Cases covered

- Message serialization/deserialization for group message types
- Basic filtering logic is embedded within the trait implementations

Cases not covered

- Filter behavior validation — no dedicated unit tests for command filtering logic
- Event filtering verification — no tests confirming only consensus proposals pass through

- Stream implementation testing — no tests for the `Stream` trait implementation
- Edge case handling — no tests for malformed or unexpected message types
- Integration testing — no tests verifying filter behavior in full system context

5.5. Component: monad-raptorcast-secondary

Description

The raptorcast-secondary component is responsible for organizing full nodes into groups so that when a full node receives a chunk from a validator, it can rebroadcast that chunk to other full nodes in its group. This enables efficient distribution of raptorcast chunks beyond just validator-to-full-node communication.

The secondary instance operates in one of two modes:

1. **Publisher mode** (validators). It creates and manages full-node groups by inviting random subsets of full nodes to join groups for specific round intervals. It uses a scheduling algorithm that balances group formation timing with invite time-outs and round deadlines.
2. **Client mode** (full nodes). It joins groups and participates in rebroadcasting. Full nodes receive group invitations from validators and decide whether to accept based on bandwidth capacity, timing constraints, and overlap-prevention rules.

Group formation protocol

There are several message types.

- **`PrepareGroup`** — a validator invitation to join a group for rounds `[start_round, end_round)`. It contains group parameters (size, duration, round span) and specifies bandwidth requirements.
- **`PrepareGroupResponse`** — full node acceptance/rejection. It includes bandwidth-availability confirmation and validates timing and resource constraints.
- **`ConfirmGroup`** — finalizes group membership. It distributes a complete member list to all participants and establishes rebroadcasting responsibilities.

Client (full node) behavior

Regarding group acceptance criteria, full nodes evaluate invitations based on the following:

- **Timing constraints.** Groups must be within an acceptable future window — the minimum is at least one round in future (prevents immediate groups) and the maximum is within the configured maximum future rounds.
- **Bandwidth budget.** Groups are accepted only if the linear cost calculation indicates sufficient bandwidth capacity remains after accounting for existing commitments.

- **Overlapping groups.** Validators cannot create overlapping groups.
- **Heartbeat mechanism.** There are relaxed constraints if no recent proposals are within configured time-out.

Publisher (validator) behavior

These are the main components of group scheduling:

- Group size — configurable maximum number of full nodes per group
- Duration — configurable round span for group lifetime
- Lookahead — start inviting with configurable advance rounds
- Formation deadline — must complete with configured rounds before group starts

These are the main components of the full-node selection strategy:

- Prioritized nodes — always invited trusted full nodes
- Random selection — additional nodes selected randomly
- Retry logic — configurable maximum rounds to wait for response before trying others

Rebroadcasting mechanism

When a full node receives chunks as part of a group, it

1. validates that the chunk belongs to the group's validator,
2. forwards to other group members (excluding the sender),
3. uses the same raptor encoding for reliability, and
4. applies the configured redundancy factor for full nodes.

State management

Publisher-state management involves the following:

- Active groups per round span
- Pending invitations tracking
- Formation progress monitoring
- Bandwidth allocation per group

Client-state management involves the following:

- Accepted groups registry
- Bandwidth-commitment tracking
- Round-based group activation
- Overlap detection mechanism using `IntervalMap`

Invariants

- Full nodes must not accept overlapping group invitations from the same validator
- Group duration must fall within the configured future distance bounds (`invite_future_dist_min` to `invite_future_dist_max`).
- Total bandwidth commitment across all accepted groups must not exceed `bandwidth_capacity`.
- Validators can only create one group per round interval.
- Group confirmations must match the original invitation parameters exactly.
- The `ConfirmGroup` peer count must not exceed the promised `max_group_size`.
- Full nodes must be included in the peer list of groups they are confirmed for.
- Group scheduling must respect `invite_lookahead` and `deadline_round_dist` timing constraints.
- Groups are automatically cleaned up when their round intervals expire.

Test coverage

Cases covered

Based on source code analysis, monad-raptorcast-secondary has the following test coverage:

- Group message serialization — comprehensive roundtrip testing for `PrepareGroup`, `PrepareGroupResponse`, and `ConfirmGroup` message types
- Publisher scheduling logic — 6 dedicated unit tests covering group formation timing, invite handling, and state transitions
- Message encoding/decoding — validation of RLP encoding for all group message variants
- State management — testing of publisher state transitions and group lifecycle management

Cases not covered

- Client-side group acceptance logic — no dedicated unit tests for full node invitation evaluation
- Byzantine validator behavior — no tests for malicious group invitation patterns
- Bandwidth budget validation — no tests for bandwidth capacity constraint checking
- Overlap detection accuracy — no tests for IntervalMap-based overlap prevention
- Group cleanup mechanisms — no tests for automatic group expiration and resource cleanup
- Error handling paths — limited coverage of network failure and malformed message scenarios

5.6. Component: monad-raptorcast primary

Description

The monad-raptorcast primary is the core broadcasting protocol component that handles efficient message dissemination across validators and full nodes using FEC techniques. It serves as the primary networking layer for communication in the Monad blockchain.

The component implements a message fragmentation and reconstruction system that splits large application messages (such as block proposals) into MTU-sized chunks, encodes them using a nonsystematic variant of the raptor code (RFC 5053) for redundancy, authenticates them with Merkle tree proofs, and distributes them across the network.

Key architectural features include the following:

- **Nonsystematic raptor FEC encoding/decoding** (RFC 5053 variant) with configurable redundancy
- **Merkle tree authentication** with fixed tree depth for chunk batches
- **Stake-weighted chunk distribution** ensuring economic incentives align with network responsibilities
- **Caching system** including signature cache, pending message cache, and recently decoded cache

Invariants

- All outbound chunks must be cryptographically signed by the sender using the `RaptorcastChunk` signing domain.
- Each chunk must include a valid Merkle proof that can be verified against the Merkle root in the signed header.
- The redundancy factor must be within configured bounds with different defaults for validators and full nodes.
- Chunk length must meet minimum size requirements for multichunk messages to prevent DOS attacks.
- Encoding symbol IDs must be less than `MAX_REDUNDANCY` * `num_source_symbols` to prevent algorithmic-complexity attacks.
- Message cache keys must uniquely identify messages using (`unix_ts_ms`, `author`, `app_message_hash`, `app_message_len`).
- All received chunks must be from valid epoch validators when in broadcast mode.
- Merkle tree depth must be within configured bounds.
- The total number of packets per message must not exceed the configured maximum.
- The decoded message hash must match the expected `app_message_hash` in the chunk headers.

Test coverage

Cases covered

Based on source code analysis, monad-raptorcast (primary) has comprehensive test coverage including:

- FEC encoding robustness — tests for different symbol sizes and buffer count overflow protection
- Message fragmentation — validation of large message splitting and reconstruction
- Error handling — encoder error testing for oversized messages exceeding SOURCE_SYMBOLS_MAX
- Edge case resilience — zero-sized packet handling, invalid message rejection
- Rebroadcasting accuracy — verification that valid encoded symbols are rebroadcast exactly once
- Integration testing — 5 comprehensive integration tests using real UDP sockets and multiple nodes
- UDP message building — 5 unit tests covering message construction with various parameters
- Message parsing — 2 unit tests for chunk header parsing and validation
- Raptor encoding integration — testing of underlying RFC 5053 FEC implementation

Cases not covered

- Stake-weighted distribution accuracy — no tests verifying correct validator stake-based chunk allocation
- Cache behavior under pressure — no tests for signature cache, pending message cache, or decoded cache overflow
- Merkle proof validation edge cases — limited testing of malformed proof rejection scenarios
- Byzantine chunk injection — insufficient coverage for malicious chunk validation and rejection
- Network partition recovery — no tests for behavior during connectivity loss and restoration

6. Assessment Results

During our assessment on the scoped Monad targets, we discovered 11 findings. Four critical issues were found. Two were of high impact, three were of medium impact, one was of low impact, and the remaining finding was 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.