RISE

Core Node

Security Assessment Report

Version: 2.0

November, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the RISE components in scope. The review focused solely on the security aspects of these components, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the RISE components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the RISE components in scope.

Overview

RISE is a high-performance Ethereum rollup built on the OP Stack and Reth SDK that aims to deliver +1 Giga gas/s throughput and 1-3ms transaction latency through innovations like pipelined block building, parallel execution, and custom DA integrations (Celestia/GCP). This review focused on the changes made to both the OP Stack and the Reth SDK client.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the risechain/rise and risechain/rise-optimism repositories.

The scope of this time-boxed review was strictly limited to files at risechain/rise commit 1a871bc5 and all changes between risechain/rise-optimism commits c1081e3a and 84ed5c32.

The fixes of the identified issues were assessed at risechain/rise commit 5ac51b49 and risechain/rise-optimism commit 84ed5c32.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Golang and Rust.

For the Golang components, the manual review focused on identifying issues associated with the business logic implementation of the libraries and modules. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Go runtime.

Additionally, the manual review process focused on identifying vulnerabilities related to known Golang anti-patterns and attack vectors, such as integer overflow, floating point underflow, deadlocking, race conditions, memory and CPU exhaustion attacks, and various panic scenarios including `nil` pointer dereferences, index out of bounds, and explicit panic calls.

To support the Golang components of the review, the testing team may use the following automated testing tools:

- golangci-lint: `https://golangci-lint.run/`
- vet: `https://pkg.go.dev/cmd/vet`
- errcheck: `https://github.com/kisielk/errcheck`

For the Rust components, the manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team may use the following automated testing tools:

- Clippy linting: `https://doc.rust-lang.org/stable/clippy/index.html`
- Cargo Audit: `https://github.com/RustSec/rustsec/tree/main/cargo-audit`
- Cargo Outdated: `https://github.com/kbknapp/cargo-outdated`

- Cargo Geiger: `https://github.com/rust-secure-code/cargo-geiger`
- Cargo Tarpaulin: `https://crates.io/crates/cargo-tarpaulin`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 21 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Medium: 4 issues.
- Low: 9 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the RISE components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
RISE-01	EIP-2935 Storage Reads Result In Consensus Failure	Critical	Resolved
RISE-02	Transactions That Read The Pending Blockhash May Be Spammed	High	Resolved
RISE-03	Permanent Mempool DoS Through Subpool Configuration Mismatch	High	Resolved
RISE-04	Pre-execution Is Aborted If An Attribute Tx Reads The Pending Blockhash	Medium	Closed
RISE-05	`wss` Should Be Enforced	Medium	Closed
RISE-06	Invalid Transactions Wait For Mempool Space	Medium	Resolved
RISE-07	Recently Executed Account Nonces May Be Too Low When EIP-7702 Is Active	Medium	Closed
RISE-08	User Timeout Not Respected When Submitting Transaction	Low	Resolved
RISE-09	GCS As Alt-DA Blocks Indefinitely Upon GCS Outages	Low	Closed
RISE-10	Silent Transaction Drop Due To Unhandled Broadcast Channel Lagged Error	Low	Resolved
RISE-11	Replica-Sequencer State Divergence on Failed Block Building	Low	Closed
RISE-12	The Gas Limit Of The Pre-Execution Should Be Checked	Low	Resolved
RISE-13	`recently_executed` Cache Not Cleared On Block Failures	Low	Resolved
RISE-14	`transaction_count()` May DoS The Sequencer	Low	Closed
RISE-15	Malicious Sequencer May Cause Replicas To Panic	Low	Resolved
RISE-16	Hardcoded Secrets In The Repo	Low	Closed
RISE-17	`PendingBlockState.base_fee` Is Never Written To	Informational	Resolved
RISE-18	Reth's Inner Pool Is Updated With Canonical State	Informational	Closed
RISE-19	Pre-Executed Block Count Not Reset After Error Recovery	Informational	Resolved
RISE-20	Dependency Vulnerabilities in Third-Party Crates	Informational	Resolved
RISE-21	Miscellaneous General Comments	Informational	Resolved

RISE-01	EIP-2935 Storage Reads Result In Consensus Failure	
Asset	`execution/crates/block-pipeline/src/pipeline.rs`	
Status	**Resolved:** See Resolution	
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A critical consensus failure vulnerability exists in Rise's pre-execution system that allows transactions reading the block-hash of the grandparent to access stale values. This causes the sequencer and replicas to produce divergent state roots, resulting in chain halt and consensus failure.

Rise deliberately performs EIP-2935 writes outside normal EVM execution. During block N execution, the attributes thread updates `bundle_state` but fails to update `pre_execution.cache`.

```
// pipeline.rs:527-535

execution_results.bundle_state.state.insert(
    HISTORY_STORAGE_ADDRESS,
    BundleAccount::new(..., storage, ...),
);
// @audit Writes to bundle_state only
// @audit Does NOT update pre_execution.cache
```

The cache persists across iterations. As such, when block N+1 pre-executes, it reuses the stale cache from block N's pre-execution, which never received the EIP-2935 data written by the attributes thread. The result is that when block N+1 pre-executes, transactions reading the parent block hash find the cache missing `slot[N-1]`, resulting in stale data being read.

A PoC of a modified e2e test has been shared with the dev team that confirms this finding.

Recommendations

Update the pre-execution cache after the EIP-2935 write at `pipeline.rs:535`.

Resolution

The recommendation has been implemented in PR #876. The change ensures that the pre-execution cache's EIP-2935 slots are updated with new block hashes.

RISE-02	Transactions That Read The Pending Blockhash May Be Spammed		
Asset	`execution/crates/block-pipeline/src/executor.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

If a transaction reads the pending block hash during pre-execution, an error is returned and the transaction will be skipped. However, this only occurs after the transaction has already performed some computation. This can be abused in a griefing attack by submitting gas-heavy transactions that perform large computations and read the pending block-hash afterwards. This results in lots of computation being wasted without the attacker having to pay for it.

In this scenario, an in-time block is scheduled every 12 blocks. To ensure the attacker does not have to pay for the spam transaction in the in-time block, they could replace the spam transaction with an empty transaction with the same nonce in the block before.

Additionally, by giving their spam transactions a high priority fee, the attacker can ensure their transactions are executed first in the block, potentially taking up a large part of the pre-execution time.

```rust
pub(crate) fn execute_pool_transactions(
    &self,
    evm: &mut RiseEvm<'_, impl StateProvider>,
    executed_results: &mut ExecutionResults,
    time_limit: Option,
) -> bool {
    // ...
    let res = match evm.transact(self.evm_config.tx_env(&tx)) {
        Ok(res) => res,
        Err(err) => {
            warn!(target: "rise::executor", %err, ?tx, "skipping invalid transaction");

            if matches!(
                err,
                EVMError::Database(PendingBlockCacheError::PendingBlockHashNotReady)
            ) {
                need_pending_block_hash = true; // @audit occurs if pending block hash is read
            }

            // ...

            executed_results.num_failed_txs += 1;
            continue;
        }
    };
```

Recommendations

Seeing as it is not possible to precisely determine which transactions will read the pending blockhash before executing the transactions, it may be better to revert these transactions instead of returning an error. This means the transaction will have to pay gas fees for any computations that were performed.

Resolution

A mitigation has been implemented in PR #886. The change temporarily suspends sequencing transactions from signers who have attempted to read the pending blockhash. The suspension lasts until the next in-time execution, which prevents an attacker from spamming gas-heavy transactions. However, attackers may still control several accounts to continue spamming.

The development team has indicated that a more comprehensive solution which includes monitoring and randomised scheduling of in-time blocks is being considered for future implementation, which would discourage such attacks.

RISE-03	Permanent Mempool DoS Through Subpool Configuration Mismatch		
Asset	`execution/crates/mempool/src/pool.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Certain configurations of Rise's max transaction limit and Reth's inner pool max transaction limit, allow an attacker to cause indefinite denial-of-service by exploiting the design of `wait_while_pool_full()` as a blocking mechanism. The attack seeks to fill the `basefee` and `queued` subpools, ensure the `pending` subpool is empty, all while hitting the Rise mempool `RISE_TX_POOL_SIZE_LIMIT` .

Rise's mempool is a thin wrapper around Reth's mempool. Internally, Reth maintains several subpools: namely, `pending` , `queued` , `basefee` . These pools contain transactions which are ready to execute (`pending`), transactions which either have a nonce which is too high or do not have sufficient funds (`queued`), and transactions which are awaiting a lower `basefee` .

Consider the following vulnerable configuration.

Reth Subpool Limits

```
PENDING=100_000
BASEFEE=100_000
QUEUED=100_000
```

Rise Limit

```
RISE_TX_POOL_SIZE_LIMIT=200_000
```

Attack Path:

1. **Fill `queued` and `basefee` :** Attacker sends 100k txs to `basefee` , and 100k txs to `queued` . The `queued` subpool is filled by submitting transactions with nonce gaps or insufficient balance. The `basefee` subpool is filled by submitting transactions with a low `max_fee_per_gas` .
2. **Hit Rise Limit:** No more txs can be added since the Rise limit has been hit
3. **Cleanup Deadlock**: `discard_worst()` cannot trigger because insertions are blocked at `wait_while_pool_full()` before reaching cleanup logic. `discard_worst()` is only called after successful insertions, but all insertions are blocked, creating a deadlock where the pool cannot self-heal. The mempool is permanently locked and no txs can be executed since there are no transactions `pending` .

Note 1: there are additional dangerous possible subpool configurations that can take advantage of a transient subpool overflow that occurs in Reth during `update_basefee()` . For example if `PENDING=300_000` , `BASEFEE=100_000` , `QUEUED=200_000` and `RISE_TX_POOL_SIZE_LIMIT = 300_000` .

Note 2: If the configuration is set such that `RISE_TX_POOL_SIZE_LIMIT` is larger than the sum of Reth's inner pool limits, then `wait_while_pool_full()` is non-functional. This is because Reth's eviction mechanism will work to ensure the subpools remain beneath their limits. In this configuration, the `RISE_TX_POOL_SIZE_LIMIT` would never be reached.

Root causes:

1. **Tx Parking Design -** `wait_while_pool_full()` checks `pool.size().total < threshold` before acquiring write lock for insertion (`pool.rs:144`). This blocks insertions when pool is full, assuming transactions will be mined and the pool will naturally drain. The assumption breaks when pool contains unexecutable transactions (`basefee` / `queued`) that persist indefinitely.
2. **Cleanup deadlock -** `discard_worst()` cannot trigger because insertions blocked at `wait_while_pool_full()` before cleanup logic runs. `discard_worst()` is only called after successful insertions, but insertions blocked at `wait_while_pool_full()` .

To avoid this issue the `RISE_TX_POOL_SIZE_LIMIT` should always be larger than the sum of Reth's subpool limits.

Recommendations

1. **Validate Configuration at Startup**
   Ensure `RISE_TX_POOL_SIZE_LIMIT >= sum(subpool_limits)` . Then fail with clear error if misconfigured.
2. **Consider Redesigning Blocking Mechanism**
   Current design assumes the pool drains via execution. However, this fails when unexecutable transactions (`basefee` / `queued`) accumulate.

Resolution

A fix has been implemented in PR #888. The change eliminates Rise mempool's `RISE_TX_POOL_SIZE_LIMIT` , thereby avoiding any potential configuration conflicts with Reth's inner subpool limits.

RISE-04	Pre-execution Is Aborted If An Attribute Tx Reads The Pending Blockhash		
Asset	`execution/crates/block-pipeline/src/executor.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

If an attribute transaction reads the pending blockhash during pre-execution, an error is returned. This error causes pre-execution to be aborted, meaning in-time execution is used for this block, resulting in performance degradation. An attacker could abuse this by submitting many transactions that read the pending blockhash.

```
pub(crate) fn execute_attributes_transactions(
    &self,
    attributes: &RiseOpPayloadBuilderAttributes,
    evm: &mut RiseEvm<'_, impl StateProvider>,
    executed_results: &mut ExecutionResults,
) -> Result<(), PayloadBuilderError> {
    for tx in &attributes.transactions {
        let res = match evm.transact(self.evm_config.tx_env(&tx)) {
            Ok(res) => res,
            Err(EVMError::Transaction(err)) => {
                // TODO: Should we return an error here instead?
                warn!(target: "rise::executor", %err, ?tx, "Error in sequencer transaction, skipping.");
                executed_results.num_failed_txs += 1;
                continue;
            }
            Err(err) => return Err(PayloadBuilderError::EvmExecutionError(Box::new(err))), // @audit returns error if tx reads
                ↪ pending blockhash
        };
```

```
fn run_pre_execution_thread(
    core: Arc>,
    parent_header_receiver: Receiver,
) {
    // ...

    while let Ok(parent) = parent_header_receiver.recv() {
        // ...

        if let Err(err) = core
            .executor
            .execute_attributes_transactions(attributes, &mut evm, results)
        {
            abort!(block_number, %err, "Failed to pre-execute attribute transactions, skipping pre-execution"); // @audit aborts on
                ↪ error
        }
```

Recommendations

Consider alternative approaches to executing the attribute transaction without having to abort pre-execution, for example by scheduling it for execution in the next in-time block.

Resolution

The development team has assessed that aborting pre-execution for these transactions is unlikely to be abused in practice, and the performance impact would be minimal.

The development team has indicated that attribute transactions are only included once per L1 block, not per L2 block. Moreover, attribute transactions submitted via the inbox must pay for L1 gas, making any spam costly for an attacker.

RISE-05	`wss` Should Be Enforced		
Asset	`execution/bin/rise-node/src/main.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When a replica is set up in `main()` the URL scheme for the sequencer can be either `ws` or `wss`. However, the sequencer is a highly trusted role. For example, state diffs sent by the sequencer in a shred will automatically be added to the replicas state without re-executing the underlying transactions. Therefore it is crucial that the sequencer is properly authenticated. As such, it should be enforced that `wss` is used for replicas that run in a public network.

```
if let Some(sequencer_url) = sequencer_url {
    // Validate URL is WebSocket
    let parsed_url = Url::parse(&sequencer_url).expect("Invalid sequencer URL");
    assert!(
        parsed_url.scheme() == "ws" || parsed_url.scheme() == "wss", // @audit `wss` should be enforced on public networks
        "Upstream URL must use WebSocket protocol (ws:// or wss://)"
    );
```

Recommendations

Enforce the use of `wss`.

Resolution

The development team have opted to not implement a fix at this time, and have provided the following rationale:

> *We'll indeed use `wss` for public connections, but `ws` support is still helpful for testing and for faster communication between trusted nodes in a private network.*

RISE-06	Invalid Transactions Wait For Mempool Space		
Asset	`execution/bin/rise-sequencer/src/rpc.rs`, `execution/crates/mempool/src/pool.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When the mempool is full users can spam invalid transactions and overload the sequencer's networking resources.

When a user submits a transaction and the mempool is full, Rise will keep the connection open and wait until the mempool has space for the user's transaction.

```
// file: execution/crates/mempool/src/pool.rs

async fn add_transaction(
    &self,
    origin: TransactionOrigin,
    transaction: Self::Transaction,
) -> PoolResult {
    self.wait_while_pool_full().await;
    let tx = self
        .pool
        .validator()
        .validate_transaction(origin, transaction)
        .await;
    let mut results = self.pool.add_transactions(origin, std::iter::once(tx));
    results
        .pop()
        .expect("result length is the same as the input")
}
```

However, transactions are not validated until *after* the pool has regained space. Once the mempool is full, invalid transactions will also park and consume networking resources until the mempool regains space. An attacker can take advantage of this by submitting a large volume of invalid transactions (e.g., with an invalid `chainID`) which will all require maintaining an open TCP connection as they await space.

Recommendations

Validate a transaction before waiting for available space in the mempool. Alternatively, when the mempool is full, simply drop all new transactions and return an error immediately.

Resolution

The recommendation has been implemented in PR #887. The change ensures that transactions are validated before waiting for mempool space.

RISE-07	Recently Executed Account Nonces May Be Too Low When EIP-7702 Is Active		
Asset	execution/crates/block-pipeline/src/executor.rs		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

EIP-7702 "Set Code for EOAs" is a recent addition to the EVM which allows Externally Owned Accounts (EOAs) to execute EVM bytecode. This also allows an EOA to execute the `CREATE` instruction during EVM execution, which deploys a new smart contract. The `CREATE` instruction also increments the account nonce.

Rise executes the EVM in the function `execute_pool_transactions()`. Results of the execution are accumulated in the struct `ExecutionResults`. The function updates each EOA's latest nonce by recording the nonce of the most recently executed transaction:

```
// file: execution/crates/block-pipeline/src/executor.rs

loop {
  // ...
  if let Some(pool_tx) = best_txs.next() {
    let TransactionId { sender, nonce } = pool_tx.transaction_id;
    // ... (EVM executes) ...
    executed_results.latest_nonces.insert(sender, nonce);
    // ...
  }
}
```

However, this nonce may be lower than the account's true nonce due to EIP-7702. This can result in invalid transactions (nonce too low) being selected from the mempool upon future block execution. Although the EVM will reject such a transaction, it nevertheless impacts performance.

Recommendations

Query the account's latest nonce from `evm.db()` after execution completes.

Resolution

The development team has opted to not implement a fix at this time, and have offered the following rationale:

> *Given how rare and low impact this case is, [...] we'll defer it to after implementing our own pool for more control. [...] `latest_nonces` is only used for filtering pending transactions for execution, and only [...] niche cases would be affected.*

RISE-08	User Timeout Not Respected When Submitting Transaction		
Asset	`execution/bin/rise-node/src/rpc.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Rise adds a new JSON-RPC method, `eth_sendRawTransactionSync`. This method accepts a raw transaction, as a byte string, and an optional timeout `user_timeout_ms`. The method decodes the transaction, submits it to the Rise mempool via `forward_raw_transaction_upstream()`, and awaits and returns its receipt once confirmed either in a block or a shred.

However, the user's configured timeout is only applied after the transaction has been submitted to the mempool. Because the Rise mempool blocks while full, adding a transaction to the mempool may consume a significant amount of time. This can result in users experiencing total request times greatly exceeding their configured timeout.

```rust
async fn send_raw_transaction_sync(
    &self,
    tx: Bytes,
    user_timeout_ms: Option,
) -> RpcResult {
    const MAX_TIMEOUT_MS: u64 = 2_000;

    trace!(target: "rpc::rise", "Serving eth_sendRawTransactionSync");

    let (hash, is_ready) = self.forward_raw_transaction_upstream(tx).

    // ...

    let timeout = Duration::from_millis(
        user_timeout_ms.map_or(MAX_TIMEOUT_MS, |ms| ms.min(MAX_TIMEOUT_MS)),
    );
    match self.pending_state.get_receipt(hash, timeout).await

    // ...
}
```

Recommendations

Calculate and enforce the user's timeout deadline starting from the moment the request is initiated, before submitting to the mempool.

Resolution

The recommendation has been implemented in PR #895. The change ensures that the user-configured timeout is respected from the start of the request, including the time taken to submit the transaction to the mempool.

RISE-09	GCS As Alt-DA Blocks Indefinitely Upon GCS Outages		
Asset	`rise-optimism/op-alt-da/cmd/daserver/gcs.go`, `rise-optimism/op-alt-da/daserver.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Rise adds Google Cloud Storage (GCS) as a Data Availability (DA) service. Using the default configurations available, if GCS experiences an outage, the DA service will hang until GCS becomes available again. This will, in turn, halt the publishing loop and prevent the Rise chain from making progress until the sequencer is reconfigured to use a different DA service.

DA blobs are stored in GCS buckets:

```go
// file: op-alt-da/cmd/daserver/gcs.go

func (s *GcsStore) Put(ctx context.Context, key []byte, value []byte) error {
    keyStr := hex.EncodeToString(key)
    obj := s.bucket.Object(keyStr)

        // ...

    writer := obj.NewWriter(ctx)

    if _, err := io.Copy(writer, bytes.NewReader(value)); err != nil {
        return err
    }

        // ...

    return nil
}
```

Importantly, the Google Cloud Storage package will retry indefinitely upon failure[1], and will only give up once the context `ctx` is canceled.

When run as a da-server, the context `ctx` is derived from an HTTP request object's `Context()`.

```go
// file: op-alt-da/daserver.go

if err = d.store.Put(r.Context(), comm, input); err != nil {
    d.log.Error("Failed to store commitment to the DA server", "err", err, "comm", comm)
    w.WriteHeader(http.StatusInternalServerError)
    return
}
```

Since the HTTP server is configured without request timeouts, this context is valid indefinitely, until an HTTP client closes the TCP connection.

[1] https://docs.cloud.google.com/go/docs/reference/cloud.google.com/go/storage/latest#hdr-Retrying_failed_requests

```
// file: op-alt-da/daserver.go

return &DAServer{
    log:      log,
    endpoint: endpoint,
    store:    store,
    httpServer: &http.Server{
        Addr: endpoint,
                // @audit: no timeouts configured
    },
    useGenericComm: useGenericComm,
}
```

On the client side, there are two relevant configurable options: `put-timeout` and `get-timeout`, which may be specified in `op-alt-da/cli.go`. However, they both default to indefinite timeouts.

Recommendations

Either set a timeout via the `put-timeout` or `get-timeout` options, or configure the GCS retry settings to give up and return an error after a maximum number of tries.

Resolution

The development team has opted to not implement a fix at this time as it is not intended to be used on mainnet.

RISE-10	Silent Transaction Drop Due To Unhandled Broadcast Channel Lagged Error		
Asset	`execution/crates/mempool/src/best.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The Rise best transactions iterator mishandles broadcast channel lag errors, causing newly-arrived transactions to be silently excluded from the current block without attempting recovery. This results in suboptimal block utilisation during high-throughput periods.

The broadcast channel for pending transactions has a capacity of 200 messages:

```
// reth/crates/transaction-pool/src/pool/pending.rs:60
pub fn new(ordering: T) -> Self {
    Self::with_buffer(ordering, 200)
}
```

According to Tokio's documentation, when the receiver lags behind a full channel:

```
// tokio-rs/tokio/tokio/src/sync/broadcast.rs:L1482
/// If the [`Receiver`] handle falls behind, once the channel is full, newly
/// sent values will overwrite old values. At this point, a call to [`recv`]
/// will return with `Err(TryRecvError::Lagged)` and the [`Receiver`]'s
/// internal cursor is updated to point to the oldest value still held by
/// the channel. A subsequent call to [`try_recv`] will return this value
/// **unless** it has been since overwritten. If there are no values to
/// receive, `Err(TryRecvError::Empty)` is returned.
```

However, Rise treats `Lagged` errors identically to `Empty`, missing the opportunity to recover:

```
// execution/crates/mempool/src/best.rs:139-145
fn try_add_next_pending_transaction_from(
    &mut self,
    pending_transaction_rx: &mut broadcast::Receiver,
) -> bool {
    let Ok(tx) = pending_transaction_rx.try_recv() else {
        return false;  // @audit - treats Lagged same as Empty - no recovery attempted
    };
    // ...
}
```

During high transaction throughput (>200 txs arriving during block execution):

1. Block execution begins with all existing pending transactions loaded
2. Rapid new transaction arrivals (>200) overflow the broadcast channel
3. `try_recv()` returns `Err(TryRecvError::Lagged(N))`
4. `N` amount of transactions are dropped and will not be included in the current block

Recommendations

Consider overriding Reth's default `PendingPool::new()` to use `PendingPool::with_buffer(ordering, capacity)` with a significantly larger, configurable channel capacity (e.g., 10,000) via `RISE_PENDING_TX_CHANNEL_CAPACITY` environment variable.

Resolution

The development team have indicated that when deployed the execution layer client runs with the command-line option `--txpool.max-new-pending-txs-notifications 100000` configured. This setting increases the broadcast channel capacity to 100,000 messages (from the default 200), which should mitigate the issue in practice.

RISE-11	Replica-Sequencer State Divergence on Failed Block Building		
Asset	`execution/crates/block-pipeline/src/pipeline.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When a block fails during finalisation (after pre-execution completes), shreds that were broadcast during pre-execution remain in replicas' pending state without being invalidated. This causes replicas to serve stale pending state until the next successful block is canonicalised, creating a window where RPC queries return incorrect data.

Scenario A: When dropping pre-execution, replica state is not cleaned. By this point the Rise system has already sent the shreds, so until shreds from the in-time block are sent, there is a stale state within the replica that appears in UX and can be relied on incorrectly by users.

Scenario B: An L1 reorg, a root calculation failure (lines [**`654-658`**]), withdrawal root calculation failure (lines [**`640-642`**]), or any other unanticipated failure after the shred was sent, also causes the replica's state to temporarily diverge from the sequencer.

Flow:

1. Pre-execution thread executes transactions.
2. Shreds are broadcast to replicas.
3. Replicas update their pending state based on received shreds.
4. Block finalisation fails (various points between lines [**`611-656`**]).
5. No rollback/invalidation message is sent to replicas.
6. The stale state persists in replicas until the next successful block or shred.

Recommendations

1. When pre-execution is dropped, consider explicitly invalidating the shreds that were sent to replicas.
2. Handle any failures that occur after sending shreds but before block finalisation.

Resolution

The development team has opted to not implement a fix at this time, and have offered the following rationale:

> *We acknowledge the concern and agree that we can improve it by sending a dedicated message type that rolls back shreds. However, we currently deprioritise it, as reverted shreds are very rare in practice; the new message can still fail to reach all nodes, and it's the same as any reorg that can happen to unsafe confirmations. We plan to revisit this in the future when we overhaul the replica-sequencer communication protocol.*

RISE-12	The Gas Limit Of The Pre-Execution Should Be Checked		
Asset	`execution/crates/block-pipeline/src/pipeline.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When deciding if the pre-execution should be dropped in `process_attributes()`, the gas limit used during pre-execution should also be checked to be smaller or equal than the gas limit request in the attributes. If the gas limit during pre-execution was higher than the requested gas limit, the pre-executed block may be larger than the gas limit and an invalid block would be produced. This could occur if the gas limit was lowered between blocks.

```rust
let dropping_pre_execution =
        // Empty or reorg'd pre-execution. Cannot check hash as the parent hash is unknown for pre-execution.
        pre_execution.block_number != block_number
        // This check could be tigher, by asserting that `pre_attributes` and `attributes.current` must be
        // "equivalent" to use the pre-executed results. It would be easier once we remove the L1 Block
        // transaction for in-between L2 blocks. That way we can skip `id` and `parent` and match the rest.
        // Alternatively, replace EngineAPI with something that is much tighter.
        || pre_attributes.prev_randao() != attributes.current.prev_randao()
        // Note: This re-executes L1 transactions for `no_tx_pool`, when we only need to greedily drop mempool
        // transactions if the pre-executed L1 transactions are already valid.
        || attributes.current.no_tx_pool;
```

Additionally, it should be checked that the `eip1559_params` is equal between the pre-execution and the attributes. Although it appears `eip1559_params` is not currently used.

Recommendations

Ensure fields such as `gas_limit` and `eip1559_params` are consistent between the attributes and what was used during pre-execution.

Resolution

The recommendation has been implemented in PR #896. The change adds checks to ensure the gas limit and EIP-1559 parameters used during pre-execution match those specified in the attributes before accepting pre-executed results.

RISE-13	`recently_executed` Cache Not Cleared On Block Failures		
Asset	`execution/crates/block-pipeline/src/pipeline.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Before pre-execution is started for a new block, the "recently executed" transaction nonces are marked in the mempool cache. However, if block finalisation fails anywhere in lines [**614-729**], the mempool cache has nonces marked as executed from a block that never made it to the canonical chain.

Transactions included in the failed block are prevented from being included in the subsequent block. The cache is only reset when the next block finalises.

```
// pipeline.rs:607-610

self.core
  .executor
  .pool
  .push_recently_executed(block_number, latest_nonces); // @audit setting nonces to recently_executed, problem if block failure
        ↪   occurs after
```

Recommendations

Consider clearing the “recently executed” cache if sealing the block fails.

Resolution

The recommendation has been implemented in PR #897. The change ensures that the “recently executed” cache is cleared in the event of a reorg, which includes block finalisation failures.

RISE-14	`transaction_count()` May DoS The Sequencer		
Asset	`execution/bin/rise-node/src/rpc.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When an RPC call to `getTransactionCount` occurs, the replica will execute `transaction_count()`. This function forwards the request to its sequencer which, in turn, forwards it again until the actual single sequencer is reached. The reason for this is that the single sequencer's mempool is relied on as a single point of truth to get the user's transaction count. However, given that the single sequencer has to process all `getTransactionCount` from all the replicas in the network, this forms a large DoS vector.

```rust
async fn transaction_count(
    &self,
    address: Address,
    block_number: Option,
) -> RpcResult {
    trace!(target: "rpc::rise", ?address, ?block_number, "Serving eth_getTransactionCount");

    if should_use_pending_state(block_number) {
        // The sequencer's mempool is the source of truth for pending transactions.
        // Without this fallback, users get "nonce too low" errors when load balancers route
        // their `eth_getTransactionCount` to a replica that hasn't received their previous
        // transactions yet (common in high-traffic scenarios).
        // TODO: Optimize this when forwarding traffic becomes high enough to stress the sequencer.
        // Note that shreds can never be fast enough by design - they arrive post-execution while pending
        // nonces should reflect mempool state at transaction arrival. There's an inherent race
        // condition where new transactions arrive before previous shreds reach replicas.
        // The proper fix is improving transaction propagation between replicas, not relying on shreds.
        if let Ok(nonce) = self
            .sequencer_client
            .request("eth_getTransactionCount", (address, BlockId::pending())) // @audit forwards the request
            .await
        {
            trace!(target: "rpc::rise", ?address, ?nonce, "Got pending nonce from sequencer");
            return Ok(nonce);
        }
    }

    Ok(self.api.transaction_count(address, block_number).await?)
}
```

Recommendations

Refactor `transaction_count()` such that not all requests have to be handled by the single sequencer.

Resolution

The development team has opted to not implement a fix at this time, and have offered the following rationale:

We acknowledge the concern and have a plan to completely redesign mempool and transaction validation. However, that will take time. For now, to protect the Sequencer, we will rely on:

- *RPC rate limiting.*
- *The fact that most Apps cache and count nonce locally during a "session", as a nonce round-trip per transaction is unnecessarily slow.*
- *More micro-optimisations for the RPC handling.*

RISE-15	Malicious Sequencer May Cause Replicas To Panic		
Asset	`execution/crates/primitives/src/pending_state.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When a shred is received in `handle_shred()`, any accounts with code changes will have their new bytecode parsed. However `Bytecode::new_raw()` panics if the bytecode does not have the correct format. Seeing as the bytecode was already verified by the sequencer during transaction execution, this is unlikely to occur. However, a malicious sequencer can still send incorrect bytecode to replicas, causing them to panic.

```
for (address, changed_account) in &shred.state_changes {
        let new_code = changed_account.new_code.as_ref().map(|new_code| {
            // Warning: This annoyingly can panic. Shipping shred verification helps prevent
            // malicious shred attacks, but ultimately, upstream should return a `Result`.
            // Replacing `revm` with Parallel EVM will help.
            let code = Arc::new(Bytecode::new_raw(new_code.clone())); // @audit panics on incorrect bytecode
            let code_hash = code.hash_slow();
            pending_block.contracts.insert(code_hash, Arc::clone(&code));
            (code_hash, code)
        });
```

Given that a replica has existing trust assumptions towards the sequencer, this issue is rated as low severity.

Recommendations

Use `Bytecode::new_raw_checked()` instead, which returns an error on failure.

Resolution

The recommendation has been implemented in PR #900. The change replaces the use of `Bytecode::new_raw()` with `Bytecode::new_raw_checked()`, and treats any bytecode parsing errors as “empty” bytecode, preventing panics.

RISE-16	Hardcoded Secrets In The Repo		
Asset	`rise-optimism/ops-bedrock/docker-compose.yml`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Several files in the repository contain hardcoded secret keys, these include:

- `rise-optimism/ops-bedrock/docker-compose.yml:310`
- `rise-optimism/ops-bedrock/p2p-node-key.txt`
- `rise-optimism/ops-bedrock/test-jwt-secret.txt`

Secret keys should never be included in a repository. Even if they are removed, they may still be accessed through the commit history by anyone if the repository is ever made open-source.

Recommendations

Rotate these secret keys and remove them from the repository.

Resolution

The development team have indicated that these secrets are only used for development purposes, and are not used in production environments.

RISE-17	`PendingBlockState.base_fee` Is Never Written To
Asset	`execution/crates/primitives/src/pending_state.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `base_fee` field in a `PendingBlockState` is never written to, and as such will always be zero. This means that `get_transaction()` will return an incorrect effective gas price and `build_rpc_receipt()` will return an incorrect base fee.

```
pub fn get_transaction(&self, tx_hash: &B256) -> Option {
    self.find_descending(|block| {
        let idx = *block.tx_ids.get(tx_hash)?;
        let tx = block.transactions.get(idx)?.clone();

        let base_fee = block.base_fee; // @audit base fee will be zero here
        let effective_gas_price = if tx.is_deposit() {
            0
        } else {
            tx.effective_tip_per_gas(base_fee).unwrap_or_default() + base_fee as u128
        };

        // ...
```

```
fn build_rpc_receipt(
    &self,
    block: &PendingBlockState,
    idx: usize,
    tx: &Recovered,
    receipt: &OpReceipt,
) -> Option {
    let tx_hash = tx.tx_hash();
    let meta = TransactionMeta {
        tx_hash,
        index: idx as u64,
        block_hash: B256::ZERO, // Pending block has zero hash by definition
        block_number: block.block_number,
        base_fee: Some(block.base_fee), // @audit base fee will be zero here
        excess_blob_gas: Some(0),
        timestamp: block.timestamp,
    };

    //...
```

Recommendations

Consider setting `base_fee` to the correct value.

Resolution

The recommendation has been implemented in PR #858. The change ensures that the `base_fee` field is correctly set when a new pending block state is created.

RISE-18	Reth's Inner Pool Is Updated With Canonical State
Asset	`execution/crates/mempool/src/pool.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `on_canonical_state_change()` function updates the inner reth pool with new state when a new canonical block is received. As such, the inner reth pool is only updated with canonical blocks, not pending state. This is in contrast with the rest of the mempool which does use pending state. For example, when adding a transaction to the pool, it is validated with respect to the pending state. As a result of this, the reth pool may hold back transactions that would already be considered valid in the pending state but are not valid yet in canonical state, causing a delay in the execution of the transaction. For example:

1. A transaction is submitted that is currently not valid because Alice's balance is too low.
2. Alice receives some funds from Bob, this is reflected in pending state, but not yet in canonical state.
3. As such, Reth's pool still considers Alice's tx as invalid.
4. Only when the canonical state is updated and `on_canonical_state_change()` is called, will reth consider the tx valid and move it to the pending pool where it can be picked up by Rise's pool.

If instead, the transaction was submitted right after step 2, it would be considered valid and could be executed right away.

```
fn on_canonical_state_change(&self, update: CanonicalStateUpdate<'_, B>) {
    let update_kind = update.update_kind;
    let block_number = update.number();

    self.pool.on_canonical_state_change(update); // @audit reths inner pool is updated here
    self.latest_canonical_update_tx.send_replace(());

    // ...
```

Recommendations

Consider updating the inner pool with pending state instead of canonical state.

Resolution

The development team has opted to not implement a fix at this time, and have offered the following rationale:

> *We acknowledge the concern and indeed have a plan to re-design the mempool accordingly — to remove transactions right from transaction execution time to improve consistency and significantly reduce friction.*

RISE-19	Pre-Executed Block Count Not Reset After Error Recovery
Asset	`execution/crates/block-pipeline/src/pipeline.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The struct `PreExecution` tracks the current in-progress state during pre-execution of a block. It can only be accessed via a `MutexGuard`, which ensures exclusive access in critical sections. The function `run_pre_execution_thread()` contains a main execution loop for the pre-execution subsystem. Within the loop, the mutex is acquired and pre-execution is performed for a single block. The number of pre-executed blocks is recorded in the variable `pre_executed_blocks`, and this is incremented upon each completion of the loop. Periodically, after a certain number of blocks are pre-executed, an in-time execution is scheduled, the pre-execution state is reset, and the value of `pre_executed_blocks` is reset to zero. This reset also occurs upon hardfork activation and in response to various errors.

Upon acquiring the `MutexGuard` it is also possible that the mutex is "poisoned" -- i.e., a thread which previously held the mutex resulted in a panic. When a "poisoned" state is encountered, the execution state is also reset:

```rust
// file: execution/crates/block-pipeline/src/pipeline.rs

let mut pre_execution = match core.pre_execution.lock() {
    Ok(pre_execution) => pre_execution,
    Err(mut e) => {
        warn!(target: "rise::pipeline", "pre-execution lock poisoned, clearing and skipping pre-execution (no valid attributes)");
        e.get_mut().clear();
        core.pre_execution.clear_poison();
        continue;
    }
};
```

However, the variable `pre_executed_blocks` is not reset to zero. This creates a state inconsistency between the object `pre_execution` and the counter `pre_executed_blocks`. This can cause premature scheduling of in-time executions.

Recommendations

Reset the counter to zero upon clearing mutex poison:

```rust
e.get_mut().clear();
pre_executed_blocks = 0;
core.pre_execution.clear_poison();
```

Resolution

The recommendation has been implemented in PR #901. The change ensures that the counter `pre_executed_blocks` is reset to zero when the pre-execution state is cleared due to mutex poisoning.

RISE-20	Dependency Vulnerabilities in Third-Party Crates
Asset	`*.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Static analysis identified two unpatched vulnerabilities in transitive dependencies of the Rise execution layer:

RUSTSEC-2025-0073: DoS Vulnerability in alloy-dyn-abi

The codebase depends on `alloy-dyn-abi v1.4.0`, which contains an uncaught panic vulnerability in the `TypedData::eip712_signing_hash()` function. The vulnerability is triggered when malformed or empty input is passed to the `TypedData` structure, causing the `eip712::Resolver::encode_type` function to panic.

Dependency Chain:

```
alloy-dyn-abi v1.4.0
  └── reth-rpc-eth-api v1.8.2 (git: paradigmxyz/reth@8effbf2)
      └── reth-rpc v1.8.2
          └── rise-primitives v0.1.0
              ├── rise-node v0.1.0
              └── rise-sequencer v0.1.0
```

Exploitation Scenario: While the vulnerability exists in the dependency, the primary attack vector (`eth_signTypedData` RPC endpoint) requires configured account signers to be exploitable. The Rise node configuration does not appear to enable signers in production deployments. However, development and test environments using the `--dev` flag or manual signer configuration would be vulnerable to denial-of-service attacks via this endpoint.

RUSTSEC-2025-0055: Log Poisoning in tracing-subscriber

The codebase transitively depends on `tracing-subscriber v0.2.25` through the `revm-precompile` dependency chain. This version is vulnerable to log poisoning attacks where user-controlled input logged by the application can inject ANSI escape sequences, potentially manipulating terminal output or compromising log integrity in monitoring systems.

Dependency Chain:

```
tracing-subscriber v0.2.25
  └── ark-relations v0.5.1
      └── ark-r1cs-std v0.5.0
          └── ark-bn254 v0.5.0
              └── revm-precompile v27.0.0
                  └── revm v29.0.1
                      └── rise-block-pipeline v0.1.0
```

This vulnerability has lower immediate impact as it affects log integrity rather than node availability or consensus.

Additional Findings:

Unmaintained Dependencies:

- `derivative v2.2.0` (RUSTSEC-2024-0388) - Used for deriving trait implementations
- `paste v1.0.15` (RUSTSEC-2024-0436) - Used for token pasting in macros

Yanked Crate (Resolved):

- `alloy-op-evm v0.21.2` was yanked due to bugs, successfully updated to v0.21.3

Recommendations

1. **Update alloy-dyn-abi (High Priority):** Update the Reth dependency to a commit that includes `alloy-dyn-abi >= v1.4.1`, which includes validation to prevent empty element access. This fix was published on 2025-10-15.
2. **Update tracing-subscriber (Medium Priority):** When upgrading revm dependencies, ensure `tracing-subscriber` is updated to `>= v0.3.20` to address the log poisoning vulnerability.
3. **Disable Signing Endpoints in Production:** Explicitly document that `eth_signTypedData` and related signing endpoints should remain disabled in production configurations. Consider adding explicit endpoint filtering to ensure these are not exposed.

Resolution

`alloy-dyn-abi` has been updated to version 1.4.1 in PR #866.

`tracing-subscriber` has been updated to version 0.3.20 in PR #902.

Signing endpoints have been disabled in PR #903.

RISE-21	Miscellaneous General Comments
Asset	All files
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **TODO Comments In Production Code**
   Related Asset(s): *.go, *.rs
   The testing team discovered TODO comments in the following code:
   (a) `execution/bin/rise-node/src/rpc.rs` lines `127`, `156`
   (b) `execution/bin/rise-sequencer/src/rpc.rs` line `45`
   (c) `execution/crates/block-pipeline/src/executor.rs` lines `243`, `361`, `470`, `519`, `535`, `541`, `592`
   (d) `execution/crates/block-pipeline/src/payload.rs` lines `180`, `246`
   (e) `execution/crates/block-pipeline/src/pipeline.rs` lines `200`, `220`, `302`, `398`, `404`, `479`, `551`, `621`
   (f) `execution/crates/block-pipeline/tests/pending_blockhash.rs` line `31`
   (g) `execution/crates/mempool/src/best.rs` lines `73`, `106`, `127`, `188`
   (h) `execution/crates/mempool/src/pool.rs` lines `8`, `46`
   (i) `execution/crates/mempool/src/validator.rs` lines `39`, `59`, `80`
   (j) `execution/crates/primitives/src/evm.rs` line `14`
   (k) `execution/crates/primitives/src/identity_hasher.rs` lines `1`, `16`
   (l) `execution/crates/primitives/src/pending_state.rs` lines `88`, `99`, `133`, `184`, `280`, `326`, `331`
   (m) `execution/crates/primitives/src/rpc.rs` lines `72`, `199`, `238`, `330`, `353`, `374`, `396`
   (n) `execution/crates/primitives/src/shred.rs` lines `36`, `47`, `82`
   (o) `op-node/rollup/attributes/attributes.go` line `235`
   (p) `op-node/rollup/derive/attributes.go` line `86`
   (q) `op-node/rollup/derive/deriver.go` lines `135`, `224`
   (r) `op-node/rollup/sequencing/sequencer.go` lines `33`, `516`
   (s) `op-service/eth/types.go` line `553`

   Consider resolving and removing the comments before deployment.
2. **Stale Payload Cache After Reset Causes Unnecessary Recursion In Block Building**
   Related Asset(s): rise-optimism/op-node/rollup/derive/deriver.go, rise-optimism/op-node/rollup/sequencing/sequencer.go
   Rise's parallel event processing modification causes stale payload envelopes to be cached after sequencer resets, leading to unnecessary cache invalidation and recursion in subsequent block-building cycles.

Rise modified the deriver to process events in parallel via a buffered channel to handle high TPS scenarios (10k+ TPS):

```
func (d *PipelineDeriver) OnEvent(ctx context.Context, ev event.Event) bool {
    // ...
    d.ch <- ev  // @audit - events queued in FIFO channel for async processing
    return true
}
```

This parallel processing introduces a race condition where `PayloadSuccessEvent` may be processed after a `ResetEvent` has already cleared the sequencer state.

Flow:

(a) Block N sealing queues `PayloadSuccessEvent` with Block N data
(b) L1 reorg detected (derivation thread, l1_traversal.go:69) which returns a `NewResetError`
(c) This error propagates, reaching `deriver.go`, which emits `ResetEvent`
(d) As the event channel accepts new entries in parallel, the reset event goes in as soon as its emitted
(e) `onBuildSealed` finishes and finally emits `PayloadSuccessEvent`
(f) The event channel is now -> `{[]ResetEvent,\ PayloadSuccessEvent{}]}`
(g) The reset event clears `d.latest` to `BuildingState{}` (zero value)
(h) `PayloadSuccessEvent` processed with `d.latest.Ref == (eth.L2BlockRef{})`
(i) Check `if d.latest.Ref != (eth.L2BlockRef{})` evaluates to false, continues execution
(j) `CachePayloadEnvelope()` caches stale Block N
(k) Next block cycle retrieves stale cache, triggers mismatch and recursion in `consolidateNextSafeAttributes`

Consider not setting the cache in `onPayloadSuccess` if `d.latest.Ref == eth.L2BlockRef{}`.

3. **Silent Failure Handling Obscures Critical Errors in Block Pipeline**

   Related Asset(s): execution/crates/block-pipeline/src/pipeline.rs

   The block pipeline's attributes processing thread handles all payload building failures identically, logging a warning and continuing to the next payload. This uniform error handling makes it difficult to distinguish between benign expected failures and critical system failures (thread crashes, consensus failures, block finalisation failures, state root calculation failures, etc).

   Affected Code `pipeline.rs:352-361`:

```
fn run_attributes_processing_thread(self) {
    while let Ok(new_attributes) = self.attributes_receiver.recv() {
        self.core.executor.set_stop_mempool_execution(true);

        if let Err(err) = self.process_attributes(new_attributes) {
            warn!(target: "rise::pipeline", %err, "skipping failed payload attributes");
            // All errors treated identically - just log and continue
        }
    }
}
```

   Consider categorizing errors to facilitate better monitoring and alerting.

4. **Testnet Gas Price Implementation in Production Code**

   Related Asset(s): execution/bin/rise-node/src/rpc.rs

   The RPC methods `eth_gasPrice` and `eth_maxPriorityFeePerGas` return hardcoded testnet values instead of dynamically calculated gas prices based on current network conditions. These methods are critical components of the Ethereum JSON-RPC API that wallets and clients use to estimate appropriate gas pricing for transaction submission.

```
async fn gas_price(&self) -> RpcResult {
    // We want (at least for the testnet) to be insanely cheap so
    // suggesting a small value (~1/10 of competitors) here.
    // Returning a constant is also much faster than fetching recent
    // (heavy) blocks for dynamic calculation. Regardless, once we
    // pick mempool transactions based on gas we may want to revert
    // this and use things like `--gpo.maxprice` instead.
    Ok(U256::from(100_000))
}
```

The method returns a constant value of 100,000 wei (0.0000001 gwei) regardless of network conditions

```
async fn max_priority_fee_per_gas(&self) -> RpcResult {
    // We want (at least for the testnet) to be insanely cheap so
    // suggesting zero value here.
    // Returning a constant is also much faster than fetching recent
    // (heavy) blocks for dynamic calculation. Regardless, once we
    // pick mempool transactions based on gas we may want to revert
    // this and use things like `--gpo.maxprice` instead.
    Ok(U256::ZERO)
}
```

The method returns zero, suggesting no priority fee is required.

Ensure dynamic gas price oracle is maintained

5. **EVM Busy Waits For Mempool Transactions**

   Related Asset(s): execution/crates/block-pipeline/src/executor.rs, execution/crates/mempool/src/best.rs

   The function `execute_pool_transactions()` contains a loop which executes transactions as they arrive in the mempool. The loop exits after a fixed timeout, or when signaled to exit by the flag `stop_mempool_execution`.

   While running, the loop periodically checks for available transactions in the mempool by polling the `next()` method on the `RiseBestTransactionsHandle` object, `best_txs`. The `next()` function inspects its local pool and, if empty, attempts to put another transaction into the pool.

```
// file: execution/crates/mempool/src/best.rs

pub fn next(&mut self) -> Option {
    if self.state.independent.is_empty() {
        self.state
            .try_add_next_pending_transaction_from(&mut self.pending_transaction_rx);
    }
    self.state.try_pop_best_transaction()
}
```

   If no transactions are available `next()` returns `None`.

```
// file: execution/crates/mempool/src/best.rs

loop {
    // @audit:

    if let Some(pool_tx) = best_txs.next() {
        // ...
    }
}
```

   This forms a busy-wait loop. Busy-waiting wastes CPU resources when the machine is otherwise idle.

   Use threading synchronization mechanisms to wait until either a new transaction is available or a stop-loop condition occurs.

6. **Usage of `usize`**

   Related Asset(s): execution/crates/block-pipeline/src/executor.rs

   When a new `PendingBlockCache` is created, `block_number` is casted to a `usize`. Given that Rise potentially wants to make use of zkVMs in the future, this is discouraged. zkVMs generally use 32-bit architectures meaning that `usize` is smaller compared to regular nodes using a 64-bit architecture. This would lead to issues if `block_number` is ever larger than 32-bits, since the provers would overflow while normal nodes do not, leading to chain splits. Given that this is unlikely to occur, this issue is rated as miscellaneous

```
pub fn new(inner: CachedReadsDbMut<'a, DB>, pending_block_number: Option) -> Self {
    Self {
        inner,
        pending_block_number,
        pending_eip2935_slot: pending_block_number
            .map(|block_number| U256::from(block_number as usize % HISTORY_SERVE_WINDOW)), // @audit could overflow on 32-bit
                ↪  arch (zkVMs)
    }
}
```

7. **`OnEvent` Always Returns `true`**

   Related Asset(s): rise-optimism/op-node/rollup/derive/deriver.go

   The `OnEvent` function will always return `true` regardless of the actual result of the event. This is in contrast with the previous implementation that returned `true` if the event was seen as ' processed'. Seeing as the return value is only used for tracing this is given as a miscellaneous issue.

```
func (d *PipelineDeriver) OnEvent(ctx context.Context, ev event.Event) bool {
    // RISE NOTE: OP Node handles events sequentially so a heavy event would block following
    // events for a long time. This derivation process is very slow as it needs to download
    // many blocks from DA to check for safe head. At 10k TPS, it can exceed 6 seconds even
    // for a local AltDA, leading to chain lag. We therefore does this in a separate
    // goroutine to process new events (especially block building) immediately.
    // TODO: Make sure it is safe to do this in parallel to block building. For example, if
    // the chain could recover/reorg correctly if this check fails.
    //
    // See: https://github.com/ethereum-optimism/optimism/issues/10864
    // See: https://github.com/megaeth-labs/optimism/pull/4
    d.ch <- ev
    return true
}
```

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows.

1. The development team has acknowledged the presence of TODO comments in the codebase and has committed to reviewing and addressing them as part of their ongoing development process.
2. Unresolved.
3. Unresolved.
4. Addressed in PR #857.
5. Unresolved.
6. Unresolved.
7. Unresolved.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact				
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

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