Coinbase: Kona

Security Review

Cantina Managed review by:
Haxatron, Lead Security Researcher
Mustafa Hasan, Lead Security Researcher

March 28, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Base is a secure, low-cost, builder-friendly Ethereum L2 built to bring the next billion users onchain.

From Jan 19th to Feb 9th the Cantina team conducted a review of kona on commit hash 86910c91. The team identified a total of **55** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	2	2	0
High Risk	1	1	0
Medium Risk	2	0	2
Low Risk	32	0	32
Gas Optimizations	0	0	0
Informational	18	0	18
Total	**55**	**3**	**52**

2.1 Scope

The security review had the following components in scope for kona on commit hash 86910c91:

```
├── bin/client/src/fpvm_evm
│   ├── factory.rs
│   ├── mod.rs
│   └── precompiles
│       ├── bls12_g1_add.rs
│       ├── bls12_g1_msm.rs
│       ├── bls12_g2_add.rs
│       ├── bls12_g2_msm.rs
│       ├── bls12_map_fp.rs
│       ├── bls12_map_fp2.rs
│       ├── bls12_pair.rs
│       ├── bn128_pair.rs
│       ├── ecrecover.rs
│       ├── kzg_point_eval.rs
│       ├── mod.rs
│       ├── provider.rs
│       ├── test_utils.rs
│       └── utils.rs
└── crates
    ├── proof
    │   ├── driver
    │   │   ├── Cargo.toml
    │   │   └── src
    │   │       ├── core.rs
    │   │       ├── cursor.rs
    │   │       ├── errors.rs
    │   │       ├── executor.rs
    │   │       ├── lib.rs
    │   │       ├── pipeline.rs
    │   │       └── tip.rs
    │   ├── executor
    │   │   ├── Cargo.toml
    │   │   ├── CHANGELOG.md
    │   │   ├── README.md
    │   │   ├── src
    │   │   │   ├── errors.rs
    │   │   │   ├── lib.rs
```

```
│   │   │   ├── test_utils.rs
│   │   │   ├── util.rs
│   │   │   ├── builder
│   │   │   │   ├── assemble.rs
│   │   │   │   ├── core.rs
│   │   │   │   ├── env.rs
│   │   │   │   └── mod.rs
│   │   │   └── db
│   │   │       ├── mod.rs
│   │   │       └── traits.rs
│   │   └── testdata
│   │       ├── block-26207960.tar.gz
│   │       ├── block-26207961.tar.gz
│   │       ├── block-26207962.tar.gz
│   │       ├── block-26207963.tar.gz
│   │       ├── block-26208384.tar.gz
│   │       ├── block-26208858.tar.gz
│   │       ├── block-26208927.tar.gz
│   │       └── block-26211680.tar.gz
│   ├── mpt
│   │   ├── Cargo.toml
│   │   ├── CHANGELOG.md
│   │   ├── README.md
│   │   ├── benches
│   │   │   └── trie_node.rs
│   │   └── src
│   │       ├── errors.rs
│   │       ├── lib.rs
│   │       ├── list_walker.rs
│   │       ├── node.rs
│   │       ├── noop.rs
│   │       ├── test_util.rs
│   │       ├── traits.rs
│   │       └── util.rs
│   ├── preimage
│   │   ├── Cargo.toml
│   │   ├── CHANGELOG.md
│   │   ├── README.md
│   │   └── src
│   │       ├── errors.rs
│   │       ├── hint.rs
│   │       ├── key.rs
│   │       ├── lib.rs
│   │       ├── native_channel.rs
│   │       ├── oracle.rs
│   │       └── traits.rs
│   ├── proof
│   │   ├── Cargo.toml
│   │   ├── CHANGELOG.md
│   │   ├── README.md
│   │   └── src
│   │       ├── blocking_runtime.rs
│   │       ├── boot.rs
│   │       ├── caching_oracle.rs
│   │       ├── eip2935.rs
│   │       ├── errors.rs
│   │       ├── executor.rs
│   │       ├── hint.rs
│   │       ├── lib.rs
│   │       ├── sync.rs
│   │       ├── l1
```

```
│   │       │   ├── blob_provider.rs
│   │       │   ├── chain_provider.rs
│   │       │   ├── mod.rs
│   │       │   └── pipeline.rs
│   │       └── l2
│   │           ├── chain_provider.rs
│   │           └── mod.rs
│   ├── proof-interop
│   │   ├── Cargo.toml
│   │   ├── CHANGELOG.md
│   │   ├── README.md
│   │   └── src
│   │       ├── boot.rs
│   │       ├── consolidation.rs
│   │       ├── hint.rs
│   │       ├── lib.rs
│   │       ├── pre_state.rs
│   │       └── provider.rs
│   ├── std-fpvm
│   │   ├── Cargo.toml
│   │   ├── CHANGELOG.md
│   │   ├── README.md
│   │   └── src
│   │       ├── channel.rs
│   │       ├── errors.rs
│   │       ├── io.rs
│   │       ├── lib.rs
│   │       ├── linux.rs
│   │       ├── malloc.rs
│   │       ├── tracing.rs
│   │       ├── types.rs
│   │       ├── mips64
│   │       │   ├── io.rs
│   │       │   ├── mod.rs
│   │       │   └── syscall.rs
│   │       ├── riscv64
│   │       │   ├── io.rs
│   │       │   ├── mod.rs
│   │       │   └── syscall.rs
│   │       └── traits
│   │           ├── basic.rs
│   │           └── mod.rs
│   └── std-fpvm-proc
│       ├── Cargo.toml
│       └── src/lib.rs
└── protocol/derive
    ├── Cargo.toml
    └── src
        ├── lib.rs
        ├── attributes
        │   ├── mod.rs
        │   └── stateful.rs
        ├── errors
        │   ├── attributes.rs
        │   ├── mod.rs
        │   ├── pipeline.rs
        │   ├── sources.rs
        │   └── stages.rs
        ├── metrics
        │   └── mod.rs
        ├── pipeline
```

```
│   ├── builder.rs
│   ├── core.rs
│   ├── mod.rs
│   └── types.rs
├── sources
│   ├── blob_data.rs
│   ├── blobs.rs
│   ├── calldata.rs
│   ├── ethereum.rs
│   ├── mod.rs
│   └── variant.rs
├── stages
│   ├── attributes_queue.rs
│   ├── frame_queue.rs
│   ├── l1_retrieval.rs
│   ├── mod.rs
│   ├── batch
│   │   ├── batch_provider.rs
│   │   ├── batch_queue.rs
│   │   ├── batch_stream.rs
│   │   ├── batch_validator.rs
│   │   └── mod.rs
│   ├── channel
│   │   ├── channel_assembler.rs
│   │   ├── channel_bank.rs
│   │   ├── channel_provider.rs
│   │   ├── channel_reader.rs
│   │   └── mod.rs
│   └── traversal
│       ├── indexed.rs
│       ├── mod.rs
│       └── polling.rs
├── traits
│   ├── attributes.rs
│   ├── data_sources.rs
│   ├── mod.rs
│   ├── pipeline.rs
│   ├── providers.rs
│   ├── reset.rs
│   └── stages.rs
└── types
    ├── mod.rs
    ├── results.rs
    └── signals.rs
```

3 Findings

3.1 Critical Risk

3.1.1 Attacker can trivially win a dispute game due to faulty trace extension mechanics

Severity: Critical Risk

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

Description: An attacker can trivially win a dispute game due to faulty trace extension mechanics in `kona`. Trace extension is a mechanism where if the output root derived by the derivation program has reached the initial output root proposed as the root claim of the dispute game, then the derivation program should stop at this output root.

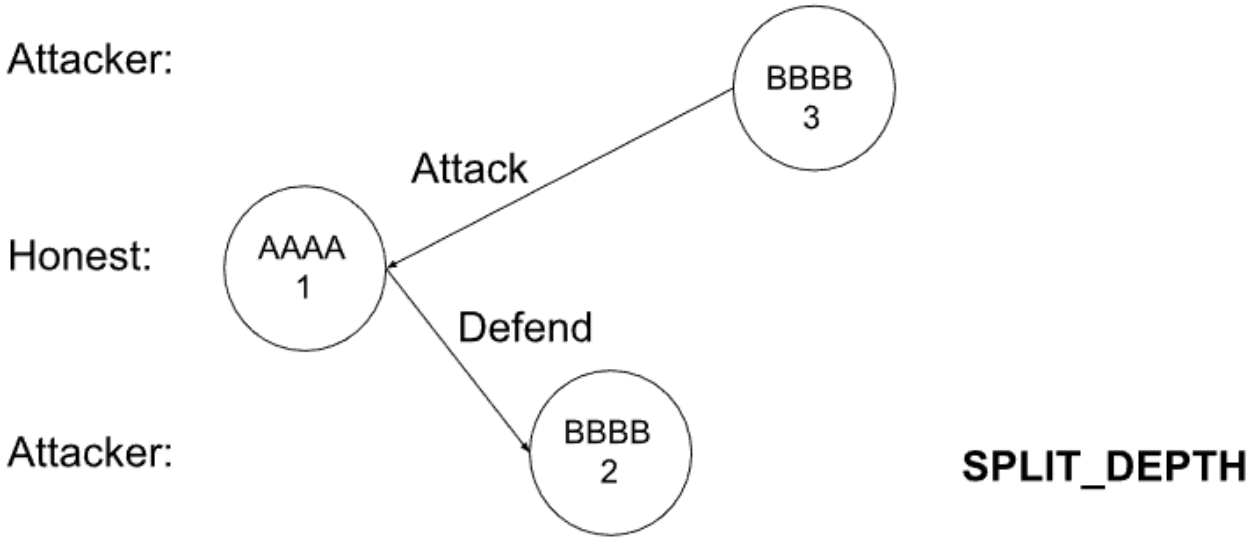
kona/bin/client/src/single.rs#L70-L93:

```
// If the claimed L2 block number is less than the safe head of the L2 chain, the claim
↪  is
// invalid.
if boot.claimed_l2_block_number < safe_head.number {
    error!(
        target: "client",
        claimed = boot.claimed_l2_block_number,
        safe = safe_head.number,
        "Claimed L2 block number is less than the safe head",
    );
    return Err(FaultProofProgramError::InvalidClaim(
        boot.agreed_l2_output_root,
        boot.claimed_l2_output_root,
    ));
}

// In the case where the agreed upon L2 output root is the same as the claimed L2 output
↪  root,
// trace extension is detected and we can skip the derivation and execution steps.
if boot.agreed_l2_output_root == boot.claimed_l2_output_root {
    info!(
        target: "client",
        "Trace extension detected. State transition is already agreed upon.",
    );
    return Ok(());
}
```

The problem with `kona` is that it fails to verify trace extension mechanism properly, specifically it only activates it when the `agreed_l2_output_root == boot.claimed_l2_output_root` when it should also verify using the claimed L2 block number (which is also capped at the block number of the root claim). An attacker can trivially trick the program into thinking it has reached the conditions of trace extension earlier than intended.

For instance, consider the actual sequence of output roots AAAA $\rightarrow$ BBBB $\rightarrow$ CCCC $\rightarrow$ ... produced by the L2 chain. Suppose we have the following subgame near the split depth:



1. The attacker proposes (either attack or defend) the output root BBBB at block 3, trace extension should not occur yet (where BBBB is not the proposed output root at the root of the dispute game).
2. Since the output root BBBB is wrong for block 3 (block 3 should be CCCC), then the honest challenger attacks with output root AAAA for block 1.
3. The attacker defends with the output root BBBB at block 2.

According to the fault dispute game mechanics, at the `SPLIT_DEPTH` the honest challenger must then show that either:

- AAAA → BBBB is invalid contradicting the attacker's defense claim in 3.
- BBBB → BBBB is invalid contradicting the attacker's claim in 1.

However `kona` resolves AAAA → BBBB as valid, and incorrectly resolves BBBB → BBBB as also valid. This prevents the honest challenger from proving the attacker's claim as invalid.

Recommendation: Only check for trace extension if the claimed L2 block number is equal to the safe head number, otherwise if the claimed L2 block number is equal to the safe head number, but trace extension is not detected, it must also reject or else the proof program would not perform trace extension accurately leading to another bypass of the dispute game.

This fix works because the FDG on-chain contract caps the L2 block number to the block number of the root claim which signals the conditions required for trace extension.

FaultDisputeGame.sol#L607-L608:

```
// Choose the minimum between the `l2BlockNumber` claim and the bisected-to L2 block
↪  number.
l2Number = l2Number < l2BlockNumber() ? l2Number : l2BlockNumber();
```

Fix:

```
  // If the claimed L2 block number is less than the safe head of the L2 chain, the claim
  ↪  is
  // invalid.
  if boot.claimed_l2_block_number < safe_head.number {
      error!(
          target: "client",
          claimed = boot.claimed_l2_block_number,
          safe = safe_head.number,
          "Claimed L2 block number is less than the safe head",
      );
      return Err(FaultProofProgramError::InvalidClaim(
          boot.agreed_l2_output_root,
          boot.claimed_l2_output_root,
      ));
```

```
  }

-  // In the case where the agreed upon L2 output root is the same as the claimed L2
↪  output root,
-  // trace extension is detected and we can skip the derivation and execution steps.
-  if boot.agreed_l2_output_root == boot.claimed_l2_output_root {
-      info!(
-          target: "client",
-          "Trace extension detected. State transition is already agreed upon.",
-      );
-      return Ok(());
-  }
+ // If the claim targets exactly the safe head block, then it must be a trace extension:
↪  the
+ // claimed output root must equal the agreed output root. Otherwise the claim is
↪  invalid.
+ if boot.claimed_l2_block_number == safe_head.number {
+     if boot.agreed_l2_output_root == boot.claimed_l2_output_root {
+         info!(
+            target: "client",
+             "Trace extension detected. State transition is already agreed upon.",
+         );
+         return Ok(());
+     }
+
+     error!(
+         target: "client",
+         claimed = boot.claimed_l2_block_number,
+         safe = safe_head.number,
+         agreed_output_root = ?boot.agreed_l2_output_root,
+         claimed_output_root = ?boot.claimed_l2_output_root,
+         "Claimed output root does not match agreed output root at safe head",
+     );
+     return Err(FaultProofProgramError::InvalidClaim(
+         boot.agreed_l2_output_root,
+         boot.claimed_l2_output_root,
+     ));
+ }
```

Coinbase: Fixed in PR 19775.

Cantina Managed: Fix verified.

3.1.2 Improper checks leading to invalid output root validation

Severity: Critical Risk

Context: sync.rs#L41

Description: Initializing the `TipCursor` with `l2_safe_head_output_root = 0x00...00` means that, if the call to `driver.advance_to_target()` in `single.rs#L138` is made with `boot.claimed_l2_block_number == tip_cursor.l2_safe_head.block_info.number` (since the check on line 72 will error out if `boot.claimed_l2_block_number < safe_head.number`), the returned `tip_cursor.l2_safe_head_output_root` will be that same zero byte root, leading to incorrect validation of an empty root hash.

Recommendation:

- Initialize the `TipCursor` with a valid `l2_safe_head_output_root`.
- Error out in case the `output_root` returned by the driver is the zero hash.

Coinbase: Fixed in PR 19775.

Cantina Managed: Fix verified.

3.2 High Risk

3.2.1 Multiple deviations in channel decompression can lead to consensus failure

Severity: High Risk

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

Description: `kona` differs from the OP stack specifications and the reference `op-node` go implementation in several ways.

crates/protocol/protocol/src/batch/reader.rs#L74-L98:

```
/// Helper method to decompress the data contained in the reader.
pub fn decompress(&mut self) -> Result<(), DecompressionError> {
    if let Some(data) = self.data.take() {
        // Peek at the data to determine the compression type.
        if data.is_empty() {
            return Err(DecompressionError::EmptyData);
        }

        let compression_type = data[0];
        if (compression_type & 0x0F) == Self::ZLIB_DEFLATE_COMPRESSION_METHOD ||
            (compression_type & 0x0F) == Self::ZLIB_RESERVED_COMPRESSION_METHOD
        {
            self.decompressed =
                decompress_to_vec_zlib(&data).map_err(|_|
                ↪  DecompressionError::ZlibError)?;

            // Check the size of the decompressed channel RLP.
            if self.decompressed.len() > self.max_rlp_bytes_per_channel {
                return Err(DecompressionError::RlpTooLarge(
                    self.decompressed.len(),
                    self.max_rlp_bytes_per_channel,
                ));
            }
        } else if compression_type == Self::CHANNEL_VERSION_BROTLI {
            self.brotli_used = true;
            self.decompressed = decompress_brotli(&data[1..],
            ↪  self.max_rlp_bytes_per_channel)?;
        } else {
            return Err(DecompressionError::UnsupportedType(compression_type));
        }
```

1. During `zlib` compression, `kona` will decompress the channel via ZLIB one-shot decompression (instead of streaming decompression) and then reject if the decompressed data exceeds `MAX_RLP_BYTES_PER_CHANNEL`. This has several implications.

- Because the entire ZLIB decompression occurs before the `max_rlp_bytes_per_channel` check, it is possible for a malicious batcher to launch a zip-bomb attack on `kona`, which when decompressed will result in out-of-memory allocation scenario and crash.
- According to the OP stack specification, if the decompressed data exceeds `MAX_RLP_BYTES_PER_CHANNEL`, the channel must be truncated to that size and not outright rejected.

According to the OP stack specification: `https://specs.optimism.io/protocol/derivation.html#channel-format`

> If the decompressed data exceeds the limit, things proceeds as though the channel contained only the first `MAX_RLP_BYTES_PER_CHANNEL` decompressed bytes.

This is not done in `kona` and the channel (and all its batches that can be decoded within the `MAX_RLP_BYTES_PER_CHANNEL` window) are rejected outright.

- Any other errors in the decompression (due to corrupted data) will result in the entire channel being rejected, while not implicit in the specifications, `op-node` differs in the sense that any batches that

can be successfully decompressed and RLP decoded before an error is encountered will be accepted.

op-node/rollup/derive/channel.go#L170-L218:

```
// BatchReader provides a function that iteratively consumes batches from the reader.
// The L1Inclusion block is also provided at creation time.
// Warning: the batch reader can read every batch-type.
// The caller of the batch-reader should filter the results.
func BatchReader(r io.Reader, maxRLPBytesPerChannel uint64, isFjord bool) (func()
↪  (*BatchData, error), error) {
    // use buffered reader so can peek the first byte
    bufReader := bufio.NewReader(r)
    compressionType, err := bufReader.Peek(1)
    if err != nil {
        return nil, err
    }

    var zr io.Reader
    var comprAlgo CompressionAlgo
    // For zlib, the last 4 bits must be either 8 or 15 (both are reserved value)
    if compressionType[0]&0x0F == ZlibCM8 || compressionType[0]&0x0F == ZlibCM15 {
        var err error
        zr, err = zlib.NewReader(bufReader)
    if err != nil {
        return nil, err
    }
        // If the bits equal to 1, then it is a brotli reader
        comprAlgo = Zlib
    } else if compressionType[0] == ChannelVersionBrotli {
        // If before Fjord, we cannot accept brotli compressed batch
        if !isFjord {
            return nil, fmt.Errorf("cannot accept brotli compressed batch before Fjord")
        }
        // discard the first byte
        _, err := bufReader.Discard(1)
        if err != nil {
            return nil, err
        }
        zr = brotli.NewReader(bufReader)
        comprAlgo = Brotli
    } else {
        return nil, fmt.Errorf("cannot distinguish the compression algo used given type
        ↪  byte %v", compressionType[0])
    }

    // Setup decompressor stage + RLP reader
    rlpReader := rlp.NewStream(zr, maxRLPBytesPerChannel)
    // Read each batch iteratively
    return func() (*BatchData, error) {
        batchData := BatchData{ComprAlgo: comprAlgo}
        if err := rlpReader.Decode(&batchData); err != nil {
            return nil, err
        }
        return &batchData, nil
    }, nil
}
```

2. Similarly, the brotli decompressor attempts to use streaming decompression. However, there are a few errors in the implementation.

crates/protocol/protocol/src/brotli.rs#L18-L76:

```
/// Decompresses the given bytes data using the Brotli decompressor implemented
/// in the [`brotli`](https://crates.io/crates/brotli) crate.
pub fn decompress_brotli(
    data: &[u8],
    max_rlp_bytes_per_channel: usize,
```

```
) -> Result<Vec<u8>, BrotliDecompressionError> {
    declare_stack_allocator_struct!(MemPool, 4096, stack);

    let mut u8_buffer = vec![0; 32 * 1024 * 1024].into_boxed_slice();
    let mut u32_buffer = vec![0; 1024 * 1024].into_boxed_slice();
    let mut hc_buffer = vec![HuffmanCode::default(); 4 * 1024 * 1024].into_boxed_slice();
    let u8_allocator = MemPool::<u8>::new_allocator(&mut u8_buffer, bzero);
    let u32_allocator = MemPool::<u32>::new_allocator(&mut u32_buffer, bzero);
    let hc_allocator = MemPool::<HuffmanCode>::new_allocator(&mut hc_buffer, bzero);
    let mut brotli_state = BrotliState::new(u8_allocator, u32_allocator, hc_allocator);

    // Setup the decompressor inputs and outputs
    let mut output = vec![0; data.len()];
    let mut available_in = data.len();
    let mut input_offset = 0;
    let mut available_out = output.len();
    let mut output_offset = 0;
    let mut written = 0;

    // Decompress the data stream until success or failure
    loop {
        match brotli::BrotliDecompressStream(
            &mut available_in,
            &mut input_offset,
            data,
            &mut available_out,
            &mut output_offset,
            &mut output,
            &mut written,
            &mut brotli_state,
        ) {
            brotli::BrotliResult::ResultSuccess => break,
            brotli::BrotliResult::NeedsMoreOutput => {
                // Resize the output buffer to double the size, following standard
                // practice for buffer resizing in streams.
                let old_len = output.len();
                let new_len = old_len * 2;

                if new_len > max_rlp_bytes_per_channel {
                    return Err(BrotliDecompressionError::BatchTooLarge);
                }

                output.resize(new_len, 0);
                available_out += old_len;
            }
            _ => break,
        }
    }

    // Truncate the output buffer to the written bytes
    output.truncate(written);

    Ok(output)
}
```

- Once again, if the decompressed data exceeds `MAX_RLP_BYTES_PER_CHANNEL`, the entire channel will be rejected instead of truncating.
- The detection of when the decompressed data would exceed `MAX_RLP_BYTES_PER_CHANNEL` is incorrect, it attempts to double the output buffer incrementally and if the output buffer exceeds `MAX_RLP_BYTES_PER_CHANNEL`, then it rejects the entire channel. Hence, if the `MAX_RLP_BYTES_PER_CHANNEL` of 10MB, and the current output buffer is currently 6MB, while the actual decompressed data should be 7.5MB. The doubling of the output buffer will exceed `MAX_RLP_BYTES_PER_CHANNEL` and the brotli decompressor will incorrectly error out even though the actual decompressed data fits well within the bounds of `MAX_RLP_BYTES_PER_CHANNEL` of 10MB.

This can be triggered by an innocent batcher that submits a channel which contains decompressed data close to, but not exceeding the size limit of `MAX_RLP_BYTES_PER_CHANNEL`.

Recommendation: Refactor the reader implementation to use streaming decompression + RLP decompression instead, making sure to limit the stream to read until first `MAX_RLP_BYTES_PER_CHANNEL` decompressed bytes.

Coinbase: Fixed in PR 455.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 Reset signals do not fully clear the derivation pipeline

Severity: Medium Risk

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

Description: On a re-org, the caller will send a reset signal to the pipeline to clear any stale data that is no longer relevant for derivation pipeline. However, the derivation pipeline is not fully cleared in two places:

1. In L1 retrieval, the data availability provider is not fully cleared, this means stale calldata or blobs that are no longer relevant can remain in the pipeline after a re-org:

crates/protocol/derive/src/stages/l1_retrieval.rs#L123-L133:

```
async fn signal(&mut self, signal: Signal) -> PipelineResult<()> {
    self.prev.signal(signal).await?;
    match signal {
        Signal::Reset(ResetSignal { l1_origin, .. }) |
        Signal::Activation(ActivationSignal { l1_origin, .. }) => {
            self.next = Some(l1_origin);
        }
        _ => {}
    }
    Ok(())
}
```

2. The payload attributes queue `DerivationPipeline.prepared` is also not cleared. This means that attributes already derived but not yet executed in the payload attributes queue will remain there after a reorg, and can be incorrectly ingested.

crates/protocol/derive/src/pipeline/core.rs#L94-L130:

```
mut s @ Signal::Reset(ResetSignal { l2_safe_head, .. }) |
mut s @ Signal::Activation(ActivationSignal { l2_safe_head, .. }) => {
    let system_config = self
        .l2_chain_provider
        .system_config_by_number(
            l2_safe_head.block_info.number,
            Arc::clone(&self.rollup_config),
        )
        .await
        .map_err(Into::into)?;
    s = s.with_system_config(system_config);
    match self.attributes.signal(s).await {
        Ok(()) => trace!(target: "pipeline", "Stages reset"),
        Err(err) => {
            if let PipelineErrorKind::Temporary(PipelineError::Eof) = err {
                trace!(target: "pipeline", "Stages reset with EOF");
            } else {
                error!(target: "pipeline", "Stage reset errored: {:?}", err);
                return Err(err);
            }
        }
    }
}
```

Although this does not affect fault dispute games, as the L1 origin is guaranteed to be canonical. This seems significant enough to warrant a fix as it can affect `kona-node`.

Recommendation: Two fixes:

1. Clear the `provider` in L1 retrieval on reset.
2. Clear the `DerivationPipeline.prepared` in payload attributes queue on reset.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 Pipeline initialization uses incorrect system config

Severity: Medium Risk

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

Description: When initializing or resetting the pipeline, the system config to use is derived from the L2 safe head.

1. The proof pipeline sends a `ResetSignal` with system config set to the L2 safe head.

kona/crates/proof/proof/src/l1/pipeline.rs#L78-L95:

```
// Reset the pipeline to populate the initial system configuration in L1 Traversal.
let l2_safe_head = *sync_start.read().l2_safe_head();
pipeline
    .signal(
        ResetSignal {
            l2_safe_head,
            l1_origin: sync_start.read().origin(),
            system_config: l2_chain_provider
                .system_config_by_number(l2_safe_head.block_info.number, cfg_for_reset)
                .await
                .ok(),
        }
        .signal(),
    )
    .await?;
```

2. The derivation pipeline overwrites the system config in the `Signal` object with the L2 safe head system config, ignoring any system config already specified in the signal.

crates/protocol/derive/src/pipeline/core.rs#L98-L105:

```
match signal {
    mut s @ (Signal::Reset(ResetSignal { l2_safe_head, .. }) |
    Signal::Activation(ActivationSignal { l2_safe_head, .. })) => {
        let system_config = self
            .l2_chain_provider
            .system_config_by_number(
                l2_safe_head.block_info.number,
                Arc::clone(&self.rollup_config),
            )
            .await
            .map_err(Into::into)?;
```

This is not correct as the pipeline starts with an L1 origin `channel_timeout` blocks away from the safe L1 origin, so the pipeline must use the system config of the L2 head with an L1 origin that is `channel_timeout` blocks away from the safe L1 origin. If the system config changes within the `channel_timeout` window, this can lead to an incorrect derivation.

For instance, if the batcher address changes within the `channel_timeout` window, then batcher transactions before the change would be incorrectly ignored as the incorrect batcher address that was changed and is now present at the `l2_safe_head.block_info.number` is incorrectly registered for blocks before the batcher address change.

Recommendation: Use the algorithm implemented by `op-node` in `initialReset`:

- Start from the reset L2 safe head.
- Look at its L1 origin (`l1_origin.number`).
- Walk back L2 parents until the candidate origin is at most `CHANNEL_TIMEOUT` L1 blocks behind the safe head's L1 origin.
- Derive system config from that L2 safe head.

op-node/rollup/derive/pipeline.go#L229-L273:

```
// initialReset does the initial reset work of finding the L1 point to rewind back to
func (dp *DerivationPipeline) initialReset(ctx context.Context, resetL2Safe
↪  eth.L2BlockRef) error {
    dp.log.Info("Rewinding derivation-pipeline L1 traversal to handle reset")

    dp.metrics.RecordPipelineReset()
    spec := rollup.NewChainSpec(dp.rollupCfg)

    // Walk back L2 chain to find the L1 origin that is old enough to start buffering
    ↪  channel data from.
    pipelineL2 := resetL2Safe
    l1Origin := resetL2Safe.L1Origin

    pipelineOrigin, err := dp.l1Fetcher.L1BlockRefByHash(ctx, l1Origin.Hash)
    if err != nil {
        return NewTemporaryError(fmt.Errorf("failed to fetch the new L1 progress: origin:
        ↪  %s; err: %w", pipelineL2.L1Origin, err))
    }

    for {
        afterL2Genesis := pipelineL2.Number > dp.rollupCfg.Genesis.L2.Number
        afterL1Genesis := pipelineL2.L1Origin.Number > dp.rollupCfg.Genesis.L1.Number
        afterChannelTimeout :=
        ↪  pipelineL2.L1Origin.Number+spec.ChannelTimeout(pipelineOrigin.Time) >
        ↪  l1Origin.Number
        if afterL2Genesis && afterL1Genesis && afterChannelTimeout {
            parent, err := dp.l2.L2BlockRefByHash(ctx, pipelineL2.ParentHash)
            if err != nil {
                return NewResetError(fmt.Errorf("failed to fetch L2 parent block %s",
                ↪  pipelineL2.ParentID()))
            }
            pipelineL2 = parent
            pipelineOrigin, err = dp.l1Fetcher.L1BlockRefByHash(ctx,
            ↪  pipelineL2.L1Origin.Hash)
            if err != nil {
                return NewTemporaryError(fmt.Errorf("failed to fetch the new L1 progress:
                ↪  origin: %s; err: %w", pipelineL2.L1Origin, err))
            }
        } else {
            break
        }
    }

    sysCfg, err := dp.l2.SystemConfigByL2Hash(ctx, pipelineL2.Hash)
    if err != nil {
        return NewTemporaryError(fmt.Errorf("failed to fetch L1 config of L2 block %s:
        ↪  %w", pipelineL2.ID(), err))
    }

    dp.origin = pipelineOrigin
    dp.resetSysConfig = sysCfg
    dp.resetL2Safe = resetL2Safe
    return nil
}
```

Coinbase: Acknowledged. We will address at a later date as we believe it to be very low risk.

Cantina Managed: Acknowledged.

3.4 Low Risk

3.4.1 Type `0x7e` batcher transactions are not supported during L1 retrieval process

Severity: Low Risk

Context: blobs.rs#L54-L68, calldata.rs#L45-L64

Description: According to the OP Stack specification, during the L1 retrieval process, the `0x7e` transaction type (deposit transaction) is supported for batcher transactions in order to support force inclusion of batcher transactions for L3s.

L1 Retrieval. In the L1 Retrieval stage, we read the block we get from the outer stage (L1 traversal), and extract data from its batcher transactions. A batcher transaction is one with the following properties:

- The `to` field is equal to the configured batcher inbox address.
- The transaction type is one of `0`, `1`, `2`, `3`, or **`0x7e` (L2 Deposited transaction type)**, to support force-inclusion of batcher transactions on nested OP Stack chains.
- The sender, as recovered from the transaction signature (`v`, `r`, and `s`), is the batcher address loaded from the system config matching the L1 block of the data.

However, the `kona` implementation does not support type `0x7e` batcher transactions and such transactions are ignored during the L1 retrieval process.

```
let (tx_kind, calldata, blob_hashes) = match &tx {
    TxEnvelope::Legacy(tx) => (tx.tx().to(), tx.tx().input.clone(), None),
    TxEnvelope::Eip2930(tx) => (tx.tx().to(), tx.tx().input.clone(), None),
    TxEnvelope::Eip1559(tx) => (tx.tx().to(), tx.tx().input.clone(), None),
    TxEnvelope::Eip4844(blob_tx_wrapper) => match blob_tx_wrapper.tx() {
        TxEip4844Variant::TxEip4844(tx) => {
            (tx.to(), tx.input.clone(), Some(tx.blob_versioned_hashes.clone()))
        }
        TxEip4844Variant::TxEip4844WithSidecar(tx) => {
            let tx = tx.tx();
            (tx.to(), tx.input.clone(), Some(tx.blob_versioned_hashes.clone()))
        }
    },
    _ => continue,
};
```

This would result in incorrect derivation particularly for OP-Stack based L3s using `kona` that settle on OP-Stack based L2, and the L3 batcher attempts to submit a deposit transaction from the L1. The deviation in the processing will result in consensus failure - `kona` will regard an output root produced by `op-node` as invalid and vice versa, resulting in incorrect resolution of a dispute game.

Recommendation: Support type `0x7e` batcher transactions during the L1 retrieval process in `kona`.

The reference `op-node` implementation:

https://github.com/ethereum-optimism/optimism/blob/develop/op-node/rollup/derive/data_source.go#L93-L118:

```
// isValidBatchTx returns true if:
//  1. the transaction type is any of Legacy, ACL, DynamicFee, Blob, or Deposit (for
↪  L3s).
//  2. the transaction has a To() address that matches the batch inbox address, and
//  3. the transaction has a valid signature from the batcher address
func isValidBatchTx(tx *types.Transaction, l1Signer types.Signer, batchInboxAddr,
↪  batcherAddr common.Address, logger log.Logger) bool {
    // For now, we want to disallow the SetCodeTx type or any future types.
    if tx.Type() > types.BlobTxType && tx.Type() != types.DepositTxType {
        return false
    }
```

```
    to := tx.To()
    if to == nil || *to != batchInboxAddr {
        return false
    }
    seqDataSubmitter, err := l1Signer.Sender(tx) // optimization: only derive sender if
    ↪  To is correct
    if err != nil {
        logger.Warn("tx in inbox with invalid signature", "hash", tx.Hash(), "err", err)
        return false
    }
    // some random L1 user might have sent a transaction to our batch inbox, ignore them
    if seqDataSubmitter != batcherAddr {
        logger.Warn("tx in inbox with unauthorized submitter", "addr", seqDataSubmitter,
        ↪  "hash", tx.Hash(), "err", err)
        return false
    }
    return true
}
```

Coinbase: Acknowledged. This doesn't strictly impact our ability to run Kona, and I don't know of any L3s running fault proofs (they usually prioritize scalability), but should probably be fixed.

L3s on the Op Stack is a very blurry notion that is not fully specked out. I'd recommend ack-ing this one and opening a github issue. This is not relevant for base.

Cantina Managed: Acknowledged.

3.4.2 Unchecked elasticity parameter in EIP-1559 decoding logic

Severity: Low Risk

Context: util.rs#L22

Description/Recommendation: Function does not check `elasticity` is non-zero. This is against the documentation which states it must be non-zero.

Coinbase: Acknowledged. Mostly a sanity check since the execution client is required to produce a non-zero `elasticity` value in the block header, but seems fine.

Cantina Managed: Acknowledged.

3.4.3 Unexpected panics on deposit-only transactions in consolidation

Severity: Low Risk

Context: consolidation.rs#L146-L149

Description: The Go implementation continues the loop if a deposit-only transaction is passed to the function re-constructing the block with deposit-only. However the function carrying out the same logic in Kona, `re_execute_deposit_only()`, reverts if the same condition occurs.

Impact Explanation: This behavior may lead to DoS.

Recommendation: Filter deposit-only transactions out and execute everything else instead of using an `assert!()` call.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future

Cantina Managed: Acknowledged.

3.4.4 Possible unexpected behavior due to vague value setting logic

Severity: Low Risk

Context: provider.rs#L130-L131

Description: `self.chain_id` is only actually updated from `None` `header_by_number()`. Also, `header_by_hash()` does not update `self.chain_id` although it is passed a `chain_id`. This may lead to unwanted behavior as there is no explicit way `self.chain_id` is set.

Recommendation: It would make more sense to implement an explicit `set_chain_id()` function like in `crates/proof/proof/src/l2/chain_provider.rs`.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.4.5 Lack of `SuperRoot` and L2 timestamp equality in `BootInfo` loading

Severity: Low Risk

Context: boot.rs#L115-L117

Description: If the `PreState` is an instance of `SuperRoot`, then its `timestamp` should be exactly equal to `claimed_l2_timestamp` (the game's timestamp) per the documentation. However such a check does not currently exist.

Recommendation: Implement a check that panics if the timestamps do not match.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.4.6 `consolidate_dependencies` will always return errors

Severity: Low Risk

Context: consolidate.rs#L117-L120

Description: This file is out of scope for the audit; however I believe it is worth mentioning that this line will always error out as `transition(None)` always returns `None`.

Recommendation: Update the logic so a valid block is passed to `transition()`.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.4.7 Incorrect logic may lead to unfinalized transition states

Severity: Low Risk

Context: pre_state.rs#L78-L110

Description: The check against the `TRANSITION_STATE_MAX_STEPS` constant is only performed if `transition_state.pending_progress.len() == transition_state.pre_state.output_roots.len()`. More steps than the maximum will be executed in case the lengths of the two vectors aren't equal before the maximum is reached. Additionally, the transition state will never be finalized as the condition `transition_state.step == TRANSITION_STATE_MAX_STEPS` will never be satisfied.

Proof of Concept: The following test case confirms the behavior:

```
#[test]
fn test_transition_state_over_max_steps() {
    const OUTPUT_ROOTS: u64 = super::TRANSITION_STATE_MAX_STEPS + 1;
    const INITIAL_STEP: u64 = super::TRANSITION_STATE_MAX_STEPS;

    let transition_state = create_test_transition_state(INITIAL_STEP, OUTPUT_ROOTS);
    let pre_state = super::PreState::TransitionState(transition_state);

    let new_pre_state_1 =
    ↪  pre_state.transition(Some(OptimisticBlock::default())).unwrap();
    let new_pre_state_2 =
    ↪  new_pre_state_1.transition(Some(OptimisticBlock::default())).unwrap();

    match new_pre_state_2 {
```

```
        super::PreState::TransitionState(state) => {
            assert!(super::TRANSITION_STATE_MAX_STEPS < state.step);
        }
        _ => panic!("Expected TransitionState"),
    }
}
```

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.4.8 Driver does not wait for the executor to be ready

Severity: Low Risk

Context: single.rs#L134

Description: The initialized `Driver` instance does not wait for the executor to be ready and `driver.advance_to_target()` is called directly after the driver initialization.

Recommendation: `driver.wait_for_executor()` should be called before `driver.advance_to_target()` to make sure the executor is in a valid state.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.9 Possible confusion on data returned from `CalldataSource.next()`

Severity: Low Risk

Context: calldata.rs#L83-L90

Description: No indicator of whether actual calldata loading happened is returned. This may lead to confusion in case the `CalldataSource` has `open` set to `true` and a `next()` call is performed with new `block_ref/batcher_address` values, since `self.calldata` will not be updated and the item returned may be confused with the first item that should be loaded from the passed `block_ref/batcher_address` combination.

Recommendation: Add a value that reflects whether new calldata was loaded to the result.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.10 Unbounded memory growth due to unremoved values

Severity: Low Risk

Context: cursor.rs#L165-L168

Description: When `advance()` is called and `self.tips.len() >= self.capacity`, the oldest value in `origins` and it's corresponding value in `tips` are removed. However, the value corresponding to `key` in `origin_infos` is not removed.

Impact Explanation: This leads to unbounded memory growth for the `origin_infos` map.

Recommendation: Remove the value from `origin_infos`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.11 Incorrect assumption may lead to panics in `PipelineCursor`

Severity: Low Risk

Context: cursor.rs#L129-L144

Description: The `tip()` function and the comment above it assume `Self::new()` would initialize the cursor with a tip corresponding to the origin passed to it, however that is never the case as `tips` is always initialized to `Default::default()`.

Recommendation: Either the comment and error message here should be updated or `Self::new()` should be changed to take a tip cursor (or similar) to properly initialize the `tips` map.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.12 `Channel Assembler` stage does not enforce strict frame ordering ingestion

Severity: Low Risk

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

Description: During the `Channel Assembler` stage (Holocene and after) does not enforce strict frame ordering ingestion as required by Holocene, a malicious batcher could submit out-of-order frames which would result in a invalid channel being accepted by `kona`, which could result in incorrect derivation and consensus failure.

In the following snippets we see `add_frame` is called in the `Channel Assembler` without specifying that strict frame ordering ingestion should be enforced.

crates/protocol/derive/src/stages/channel/channel_assembler.rs#L66-L170:

```
#[async_trait]
impl<P> ChannelReaderProvider for ChannelAssembler<P>
where
    P: NextFrameProvider + OriginAdvancer + OriginProvider + SignalReceiver + Send +
    ↪  Debug,
{
    async fn next_data(&mut self) -> PipelineResult<Option<Bytes>> {
//...
            if channel.add_frame(next_frame, origin).is_err() {
                error!(
                    target: "channel_assembler",
                    "Failed to add frame to channel (ID: {}) at L1 origin #{}",
                    hex::encode(channel.id()),
                    origin.number
                );
                return Err(PipelineError::NotEnoughData.temp());
            }
//...
    }
}
```

crates/protocol/protocol/src/channel.rs#L95-L142:

```
pub fn add_frame(
    &mut self,
    frame: Frame,
    l1_inclusion_block: BlockInfo,
) -> Result<(), ChannelError> {
    // Ensure that the frame ID is equal to the channel ID.
    if frame.id != self.id {
        return Err(ChannelError::FrameIdMismatch);
    }
    if frame.is_last && self.closed {
        return Err(ChannelError::ChannelClosed);
    }
    if self.inputs.contains_key(&frame.number) {
        return Err(ChannelError::FrameNumberExists(frame.number as usize));
    }
    if self.closed && frame.number >= self.last_frame_number {
        return Err(ChannelError::FrameBeyondEndFrame(frame.number as usize));
    }
```

```
    // Guaranteed to succeed at this point. Update the channel state.
    if frame.is_last {
        self.last_frame_number = frame.number;
        self.closed = true;

        // Prune frames with a higher number than the last frame number when we receive a
        // closing frame.
        if self.last_frame_number < self.highest_frame_number {
            self.inputs.retain(|id, frame| {
                self.estimated_size -= frame.size();
                *id < self.last_frame_number
            });
            self.highest_frame_number = self.last_frame_number;
        }
    }

    // Update the highest frame number.
    if frame.number > self.highest_frame_number {
        self.highest_frame_number = frame.number;
    }

    if self.highest_l1_inclusion_block.number < l1_inclusion_block.number {
        self.highest_l1_inclusion_block = l1_inclusion_block;
    }

    self.estimated_size += frame.size();
    self.inputs.insert(frame.number, frame);
    Ok(())
}
```

As seen above, there is no check that enforces that frames arrive into the `Channel Assembler` stage in order.

Out-of-order frames can still enter the `Channel Assembler` stage from the `Frame Queue` stage as the `Frame Queue` stage only enforces correct ordering of frames while in the queue. This means that if a frame is already dequeued into `Channel Assembler` stage, the next frame that is loaded and going to be dequeued from the `Frame Queue` stage can be out-of-order with respect to the already dequeued frame.

This scenario is acknowledged via comments in `op-node`:

op-node/rollup/derive/channel_assembler.go#L107-L115:

```
// Catches Holocene ordering rules. Note that even though the frame queue is guaranteed
 ↪  to
// only hold ordered frames in the current queue, it cannot guarantee this w.r.t. frames
// that already got dequeued. So ordering has to be checked here again.
if err := ca.channel.AddFrame(frame, origin); err != nil {
   lgr.Warn("failed to add frame to channel",
   "channel", ca.channel.ID(), "frame_channel", frame.ID,
   "frame_number", frame.FrameNumber, "err", err)
    continue // read more frames
}
```

And `op-node` enforces a `requireInOrder` switch for the `Channel.AddFrame` function. This switch is activated only after the Holocene hardfork.

op-node/rollup/derive/channel.go#L78-L80:

```
func (ch *Channel) AddFrame(frame Frame, l1InclusionBlock eth.L1BlockRef) error {
//...
    if ch.requireInOrder && int(frame.FrameNumber) != len(ch.inputs) {
        return fmt.Errorf("frame out of order, expected %d, got %d", len(ch.inputs),
         ↪  frame.FrameNumber)
    }
```

Now, to trigger the scenario, the batcher can submit at least three transactions `T1 = [F0, F1, F2]`,

`T2 = [F4, F5, F6 (is_last)]` and `T3 = [F3]` and these transactions will be queued into `Frame Queue` via separate `load_frames` calls. As a result, no frames will be pruned from the `Frame Queue` stage as they are all in-order with respect to the queue.

crates/protocol/derive/src/stages/frame_queue.rs#L108-L148:

```
    /// Loads more frames into the [`FrameQueue`].
    pub async fn load_frames(&mut self) -> PipelineResult<()> {
        // Skip loading frames if the queue is not empty.
        if !self.queue.is_empty() {
            return Ok(());
        }

        let data = match self.prev.next_data().await {
            Ok(data) => data,
            Err(e) => {
                debug!(target: "frame_queue", "Failed to retrieve data: {:?}", e);
                // SAFETY: Bubble up potential EOF error without wrapping.
                return Err(e);
            }
        };

        let Ok(frames) = Frame::parse_frames(&data.into()) else {
            // There may be more frames in the queue for the
            // pipeline to advance, so don't return an error here.
            error!(target: "frame_queue", "Failed to parse frames from data.");
            return Ok(());
        };

        // Optimistically extend the queue with the new frames.
        self.queue.extend(frames);

//...

        // Prune frames if Holocene is active.
        let origin = self.origin().ok_or(PipelineError::MissingOrigin.crit())?;
        self.prune(origin);

        Ok(())
    }
```

Then during the `Channel Assembler` stage:

- `kona` will accept `F0, F1, F2` from `T1`, and `F4, F5, F6 (is_last)` from `T2` and then `F3` from `T3`, after which the channel is considered completed and proceeds to the `Channel Reader` stage.
- `op-node` will accept `F0, F1, F2` from `T1`, discard `F4, F5, F6 (is_last)` from `T2` as they are out-of-order and then `F3` from `T3`. The channel is never completed within these set of three transactions.

The deviation in the processing will result in consensus failure - `kona` will regard an output root produced by `op-node` as invalid and vice versa, resulting in incorrect resolution of a dispute game.

Recommendation: It is recommended to enforce `requireInOrder` in the `Channel` object during Holocene just like what is done in the `op-node` implementation as highlighted.

Note that the specification also makes the incorrect assumption that since the `Frame Queue` stage guarantees that ordered and contiguous frames, the next stage in the pipeline Channel Assembler will also receive frames in-order and contiguous.

See Reading & Frame Loading in the OP Stack specification.

> The frame queue is guaranteed to hold ordered and contiguous frames, per channel. So reading and frame loading becomes simpler in the channel bank...

The specification should also be corrected to enforce that frame addition during the `Channel Assembler` stage must also be enforced to be in-order.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.13 Incorrect tracking of estimated channel size during pruning

Severity: Low Risk

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

Description: After the channel is closed (a frame with `is_last = True` is received). The estimated channel size is incorrectly tracked during pruning.

crates/protocol/protocol/src/channel.rs#L119:

```
// Prune frames with a higher number than the last frame number when we receive a
// closing frame.
if self.last_frame_number < self.highest_frame_number {
    self.inputs.retain(|id, frame| {
        self.estimated_size -= frame.size();
        *id < self.last_frame_number
    });
    self.highest_frame_number = self.last_frame_number;
}
```

Here, the closure subtracts `frame.size()` from `estimated_size` from every frame in the channel, instead of the frames that are pruned.

This can lead to incorrect tracking of `estimated_size` which can lead to memory allocation problems and consensus failure as a channel that is supposed to be discarded because it exceeds the `MAX_CHANNEL_BANK_SIZE` is not.

However, this only affects pre-Holocene, as Holocene enforces strict ordering of frames arriving into the channel in absentia of the bug in `Channel Assembler` stage does not enforce strict frame ordering ingestion.

Recommendation: We recommend fixing the closure to subtract the sizes of only the frames that are pruned from the channel nevertheless.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.14 Batch with unknown batch type causes a panic

Severity: Low Risk

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

Description: If a batch (accidentally or maliciously created by the batcher) is received with an unknown batch type (neither `SINGLE_BATCH_TYPE` nor `SPAN_BATCH_TYPE`) during the `Channel Reader` stage, the entire `kona` process will panic instead of returning a temporary error.

crates/protocol/protocol/src/batch/reader.rs#L105-L115:

```
/// Pulls out the next batch from the reader.
pub fn next_batch(&mut self, cfg: &RollupConfig) -> Option<Batch> {
    // Ensure the data is decompressed.
    self.decompress().ok()?;

    // Decompress and RLP decode the batch data, before finally decoding the batch
    ↪  itself.
    let decompressed_reader = &mut self.decompressed.as_slice()[self.cursor..].as_ref();
    let bytes = Bytes::decode(decompressed_reader).ok()?;
    let Ok(batch) = Batch::decode(&mut bytes.as_ref(), cfg) else {
        return None;
    };
```

crates/protocol/protocol/src/batch/core.rs#L38-L45:

```rust
/// Attempts to decode a batch from a reader.
pub fn decode(r: &mut &[u8], cfg: &RollupConfig) -> Result<Self, BatchDecodingError> {
    if r.is_empty() {
        return Err(BatchDecodingError::EmptyBuffer);
    }

    // Read the batch type
    let batch_type = BatchType::from(r[0]);
```

crates/protocol/protocol/src/batch/type.rs#L31-L39:

```rust
fn from(val: u8) -> Self {
    match val {
        SINGLE_BATCH_TYPE => Self::Single,
        SPAN_BATCH_TYPE => Self::Span,
        _ => panic!("Invalid batch type: {val}"),
    }
}
```

Recommendation: If the batch type is unknown an `Err` should be returned using `try_from` instead.

For reference, `op-node` implementation will ignore the batch and skip to the next channel if it encounters a malformed batch. That includes a batch with an unknown type.

op-node/rollup/derive/batch.go#L124-L143:

```go
// decodeTyped decodes a typed batchData
func (b *BatchData) decodeTyped(data []byte) error {
    if len(data) == 0 {
        return errors.New("batch too short")
    }
    var inner InnerBatchData
    switch data[0] {
    case SingularBatchType:
        inner = new(SingularBatch)
    case SpanBatchType:
        inner = new(RawSpanBatch)
    default:
        return fmt.Errorf("unrecognized batch type: %d", data[0])
    }
    if err := inner.decode(bytes.NewReader(data[1:])); err != nil {
        return err
    }
    b.inner = inner
    return nil
}
```

op-node/rollup/derive/batch.go#L104-L114:

```go
func (b *BatchData) DecodeRLP(s *rlp.Stream) error {
    if b == nil {
        return errors.New("cannot decode into nil BatchData")
    }
    v, err := s.Bytes()
    if err != nil {
        return err
    }
    return b.decodeTyped(v)
}
```

op-node/rollup/derive/channel.go#L209-L218:

```go
func BatchReader(r io.Reader, maxRLPBytesPerChannel uint64, isFjord bool) (func()
↪  (*BatchData, error), error) {
//...
    // Setup decompressor stage + RLP reader
```

```
    rlpReader := rlp.NewStream(zr, maxRLPBytesPerChannel)
    // Read each batch iteratively
    return func() (*BatchData, error) {
        batchData := BatchData{ComprAlgo: comprAlgo}
        if err := rlpReader.Decode(&batchData); err != nil {
            return nil, err
        }
        return &batchData, nil
    }, nil
}
```

op-node/rollup/derive/channel_in_reader.go#L89-L99:

```
// TODO: can batch be non nil while err == io.EOF
// This depends on the behavior of rlp.Stream
batchData, err := cr.nextBatchFn()
if err == io.EOF {
    .NextChannel()
     return nil, NotEnoughData
} else if err != nil {
     cr.log.Warn("failed to read batch from channel reader, skipping to next channel
     ↪  now", "err", err)
     cr.NextChannel()
     return nil, NotEnoughData
}
```

See Batch Format in the OP Stack specification.

> Unknown versions make the batch invalid (it must be ignored by the rollup node), as do malformed contents.

The derivation pipeline should treat malformed batches as invalid and skip past them and not panic.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.15 Brotli activation uses incorrect timestamp value

Severity: Low Risk

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

Description: During the `Channel Reader` stage, `kona` uses the individual batch timestamp value `batch.timestamp()` instead of the origin timestamp. This timestamp value is untrusted.

crates/protocol/protocol/src/batch/reader.rs#L105-L125:

```
    /// Pulls out the next batch from the reader.
    pub fn next_batch(&mut self, cfg: &RollupConfig) -> Option<Batch> {
//...
        // Confirm that brotli decompression was performed *after* the Fjord hardfork.
        if self.brotli_used && !cfg.is_fjord_active(batch.timestamp()) {
            return None;
        }
//...
    }
```

The reference `op-node` implementation uses the origin timestamp, not the batch timestamp, which can result in consensus failure.

op-node/rollup/derive/channel_in_reader.go#L57:

```
if f, err := BatchReader(bytes.NewBuffer(data),
 ↪  cr.spec.MaxRLPBytesPerChannel(cr.prev.Origin().Time),
 ↪  cr.cfg.IsFjord(cr.prev.Origin().Time)); err == nil {
```

op-node/rollup/derive/channel.go#L193-L197:

```
} else if compressionType[0] == ChannelVersionBrotli {
    // If before Fjord, we cannot accept brotli compressed batch
    if !isFjord {
        return nil, fmt.Errorf("cannot accept brotli compressed batch before Fjord")
    }
```

Even post-Fjord can be affected. A scenario where this can occur post-Fjord, post-Holocene is if the batcher submits a brotli-compressed channel `C = [B1, B2, B3]`. `B1` has an incorrect timestamp set to pre-Fjord, while `B2` and `B3` have correct timestamps. During the `Channel Reader` stage,

- `kona` will return `None` upon encountering `B1`, this causes the `Channel Reader` stage to completely discard the current channel and load the next channel from `Channel Assembler`.
- Since the origin is post-Fjord, `op-node` will not error during the `Channel Reader` stage and hence forward `B1, B2, B3` to the next stage. The incorrect timestamp is only flagged during batch validation where `B1` is discarded because of its past timestamp. `op-node` will not discard `B2` or `B3` as a validity result of `Past` does not flush a channel.

The deviation in the processing will result in consensus failure - `kona` will regard an output root produced by `op-node` as invalid and vice versa, resulting in incorrect resolution of a dispute game.

Recommendation: Use the origin timestamp for Fjord checks when determining if brotli decompression can be used, as done by `op-node`.

The OP stack specifications are a bit ambiguous on this matter. The most relevant statement notes that derivation should use the L1 origin (the block where the data is derived from) when determining when to apply the change.

See Timestamp Activation in the OP Stack specification:

> Changes to derivation are applied when it is considering data from a L1 Block whose timestamp is greater than or equal to the activation timestamp.

The specific brotli specifications noted in Brotli Channel Compression in the OP Stack specification do not prescribe based on which timestamp the change should be activated.

The specification can also be clarified on this matter.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.16 Malformed span batches with truncated contents can cause panics

Severity: Low Risk

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

Description: During batch decoding in `Channel Reader` stage, there are several places where the span batch decoding process does not account for truncated span batches and can read out-of-bounds of slice which can cause panics:

1. During prefix decoding when the size of `r` is less than 20 bytes.

crates/protocol/protocol/src/batch/prefix.rs#L48-L64:

```
/// Decodes the parent check from a reader.
pub fn decode_parent_check(&mut self, r: &mut &[u8]) -> Result<(), SpanBatchError> {
    let (parent_check, remaining) = r.split_at(20);
    let parent_check = FixedBytes::<20>::from_slice(parent_check);
    *r = remaining;
    self.parent_check = parent_check;
    Ok(())
}

/// Decodes the L1 origin check from a reader.
```

```rust
pub fn decode_l1_origin_check(&mut self, r: &mut &[u8]) -> Result<(), SpanBatchError> {
    let (l1_origin_check, remaining) = r.split_at(20);
    let l1_origin_check = FixedBytes::<20>::from_slice(l1_origin_check);
    *r = remaining;
    self.l1_origin_check = l1_origin_check;
    Ok(())
}
```

2. During `decode_tx_sigs`.

crates/protocol/protocol/src/batch/transactions.rs#L156-L157:

```rust
    /// Decode the transaction signatures from a reader (excluding `v` field).
    pub fn decode_tx_sigs(&mut self, r: &mut &[u8]) -> Result<(), SpanBatchError> {
//...
            let y_parity = y_parity_bits.get_bit(i as usize).expect("same length");
            let r_val = U256::from_be_slice(&r[..32]);
            let s_val = U256::from_be_slice(&r[32..64]);
//...
    }
```

3. During `decode_tx_tos`.

crates/protocol/protocol/src/batch/transactions.rs#L191-L202:

```rust
    /// Decode the `to` addresses of the transactions from a reader.
    pub fn decode_tx_tos(&mut self, r: &mut &[u8]) -> Result<(), SpanBatchError> {
//...
            let to = Address::from_slice(&r[..20]);
//...
            r.advance(20);
//...
    }
```

4. During `read_tx_data`, when the `payload_length_with_header` exceeds the actual length of the payload.

crates/protocol/protocol/src/utils.rs#L119:

```rust
/// Reads transaction data from a reader.
pub fn read_tx_data(r: &mut &[u8]) -> Result<(Vec<u8>, TxType), SpanBatchError> {
    // Grab the raw RLP for the transaction data from `r`. It was unaffected since we
    ↪  copied it.
    let payload_length_with_header = rlp_header.payload_length + rlp_header.length();
    let payload = r[0..payload_length_with_header].to_vec();
```

Recommendation: Return an `Err` during decoding if the specified lengths exceed the actual length of the slice.

For reference, `op-node` uses `io.readFull` to read from `r`, when the input buffer exceeds the actual length of `r` and `err` is returned:

```go
// decodeParentCheck parses data into bp.parentCheck
func (bp *spanBatchPrefix) decodeParentCheck(r *bytes.Reader) error {
    _, err := io.ReadFull(r, bp.parentCheck[:])
    if err != nil {
        return fmt.Errorf("failed to read parent check: %w", err)
    }
    return nil
}

// decodeL1OriginCheck parses data into bp.decodeL1OriginCheck
func (bp *spanBatchPrefix) decodeL1OriginCheck(r *bytes.Reader) error {
    _, err := io.ReadFull(r, bp.l1OriginCheck[:])
    if err != nil {
        return fmt.Errorf("failed to read l1 origin check: %w", err)
    }
```

```
    return nil
}
```

This `err` is bubbled up and `op-node` will ignore the batch and skip to the next channel if it encounters a malformed batch.

op-node/rollup/derive/channel_in_reader.go#L89-L99:

```
// TODO: can batch be non nil while err == io.EOF
    // This depends on the behavior of rlp.Stream
    batchData, err := cr.nextBatchFn()
    if err == io.EOF {
        .NextChannel()
         return nil, NotEnoughData
    } else if err != nil {
         cr.log.Warn("failed to read batch from channel reader, skipping to next channel
         ↪  now", "err", err)
         cr.NextChannel()
         return nil, NotEnoughData
    }
```

See Batch Format in the OP Stack specification.

> Unknown versions make the batch invalid (it must be ignored by the rollup node), as do malformed contents.

The derivation pipeline should treat malformed batches as invalid and skip past them and not panic.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.17 Incorrect hint payload may result in incorrect oracle responses

Severity: Low Risk

Context: blob_provider.rs#L43-L49

Description: The highlighted code snippet constructs the blob metadata that will be used in a hint sent to the oracle while fetching the commitment for the blob in question. The range `32..40` of bytes in the payload correspond to the index of the blob hash, and the range `40..48` points to the block timestamp. This implementation is incorrect as the former range should point to the timestamp and the former range should reference the index. The Go implementation shows the correct payload structure.

Recommendation: Reverse the payload assignment as shown in the Go implementation.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.18 No bounds on `payload_length_with_header` during span batch transaction decoding

Severity: Low Risk

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

Description: There is no bounds on `payload_length_with_header` during span batch transaction construction. A malicious batcher can submit a batch with a transaction that exceeds the length of `MaxSpanBatchElementCount` which.

- `kona` accepts:

crates/protocol/protocol/src/utils.rs#L116-L122:

```
/// Reads transaction data from a reader.
pub fn read_tx_data(r: &mut &[u8]) -> Result<(Vec<u8>, TxType), SpanBatchError> {
    //...
    let payload_length_with_header = rlp_header.payload_length + rlp_header.length();
```

```
    let payload = r[0..payload_length_with_header].to_vec();
    r.advance(payload_length_with_header);
    //...
```

- `op-node` rejects.

op-node/rollup/derive/span_batch.go#L678-L694:

```
// avoid out of memory before allocation
s := rlp.NewStream(r, MaxSpanBatchElementCount)
var txPayload []byte
kind, _, err := s.Kind()
switch {
case err != nil:
    if errors.Is(err, rlp.ErrValueTooLarge) {
        return nil, 0, ErrTooBigSpanBatchSize
    }
    return nil, 0, fmt.Errorf("failed to read tx RLP prefix: %w", err)
case kind == rlp.List:
    if txPayload, err = s.Raw(); err != nil {
        return nil, 0, fmt.Errorf("failed to read tx RLP payload: %w", err)
    }
default:
    return nil, 0, errors.New("tx RLP prefix type must be list")
}
```

Which results in consensus failure.

Recommendation: Ensure that `payload_length_with_header` does not exceed `MaxSpanBatchElementCount`.

Note that while the OP stack specifications make mention of `MAX_SPAN_BATCH_ELEMENT_COUNT`, it is ambiguous on what components of the span batch should be checked against this value.

See Span Batch Size Limits in the OP Stack specification.

> In addition to the byte limit, the number of blocks, and total transactions is limited to `MAX_SPAN_BATCH_ELEMENT_COUNT`. This does imply that the max number of transactions per block is also `MAX_SPAN_BATCH_ELEMENT_COUNT`. `MAX_SPAN_BATCH_ELEMENT_COUNT` is defined in Protocol Parameters table.

The specification could be clarified on this matter.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.19 Incorrect value passed to `is_interop_active()` leading to incorrect error handling

Severity: Low Risk

Context: core.rs#L243

Description: `is_interop_active()` expects a block timestamp but is passed a block number instead. Since the block number will always be smaller than the block timestamp, this will lead to never detecting the `EndOfSource` error and always continuing the loop. In case interop is actually enabled, the `match` block in `bin/client/src/interop/transition.rs` will branch to the `Ok()` state after the function successfully returns, constructing an invalid optimistic block.

Recommendation: Pass `block_info.timestamp` to the function call instead of `block_info.number`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.20 Incorrect padding of span batch bits when data is malformed

Severity: Low Risk

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

Description: When decoding span batch bits, if `b.len() < buffer_len` due to malformed data (truncated batches), the function will left-pad with zeroes to a multiple of 8.

crates/protocol/protocol/src/batch/bits.rs#L25-L47:

```
/// Decodes a standard span-batch bitlist from a reader.
/// The bitlist is encoded as big-endian integer, left-padded with zeroes to a multiple
↪  of 8
/// bits. The encoded bitlist cannot be longer than `bit_length`.
pub fn decode(b: &mut &[u8], bit_length: usize) -> Result<Self, SpanBatchError> {
    let buffer_len = bit_length / 8 + if !bit_length.is_multiple_of(8) { 1 } else { 0 };
    let bits = if b.len() < buffer_len {
        let mut bits = vec![0; buffer_len];
        bits[..b.len()].copy_from_slice(b);
        b.advance(b.len());
        bits
    } else {
        let v = b[..buffer_len].to_vec();
        b.advance(buffer_len);
        v
    };
    let sb_bits = Self(bits);

    if sb_bits.bit_len() > bit_length {
        return Err(SpanBatchError::BitfieldTooLong);
    }

    Ok(sb_bits)
}
```

This is not what `op-node` does, instead it errors, if it cannot read the entire buffer from IO.

op-node/rollup/derive/span_batch_util.go#L21-L24:

```
buf := make([]byte, bufLen)
_, err := io.ReadFull(r, buf)
if err != nil {
    return nil, fmt.Errorf("failed to read bits: %w", err)
}
```

The specification does not make any recommendation on this matter. However, although `kona` is incorrect the truncated batch will fail in other parts of the span batch decoding process as there would be no more RLP data left to decode.

Recommendation: Do not pad if there is not enough data, instead error out like what `op-node` does. Specification can also be improved on this matter.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.21 Incorrect `L2AccountProof` hinting (deviation from specs)

Severity: Low Risk

Context: chain_provider.rs#L236-L244

Description: Contrary to the documentation and the OP program implementation, the `L2AccountProof` hint uses an 8 byte block number instead of a 32 byte block hash.

The impact is low because the Kona host also deviates from the specifications so this does not break derivation.

Recommendation: Follow the specification and OP program implementation.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.22 Managed traversal does not reset the pipeline for Holocene activation

Severity: Low Risk

Context: indexed.rs#L63-L118, polling.rs#L138-L142

Description: The following logic is performed in the `advance_origin()` function when polling traversal is used:

```
let prev_block_holocene = self.rollup_config.is_holocene_active(block.timestamp);
let next_block_holocene =
↪  self.rollup_config.is_holocene_active(next_l1_origin.timestamp);

// Update the block origin regardless of if a holocene activation is required.
self.update_origin(next_l1_origin);

//...

// If the prev block is not holocene, but the next is, we need to flag this
// so the pipeline driver will reset the pipeline for holocene activation.
if !prev_block_holocene && next_block_holocene {
    return Err(ResetError::HoloceneActivation.reset());
}
```

However the same logic is not present when managed (indexed) traversal is in use. This means that clients using indexed traversal will not reset their derivation pipelines once Holocene activation is detected, leading to continued operation without the necessary system config updates related to the Holocene upgrade.

Recommendation: Detect the Holocene upgrade like in the polling traversal logic and return a `PipelineErrorKind::Reset()` in case Holocene is to activate.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.23 Channels with u16::MAX frames will never be ready

Severity: Low Risk

Context: channel.rs#L58, channel.rs#L155-L174, channel.rs#L162-L164

Description: In a channel, the `last_frame_number` attribute is a `u16` type so the maximum value it can hold is `65535`. In the `is_ready()` function, the highlighted check will fail as `self.last_frame_number + 1` will wrap around to zero since overflow checks are disabled, in case `last_frame_number` goes up to the maximum `u16` value.

Assuming channels can hold as much frames, this behavior would result in breaking channel readiness as sufficiently large channels will never be ready for processing, possibly affecting derivation.

Recommendation: Use a bigger value for `last_frame_number`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.24 Overallocation while returning channel data may result in OOM

Severity: Low Risk

Context: channel.rs#L150-L152, channel.rs#L182-L194, frame.rs#L261-L264

Description: A channel's `self.size()` function returns the `estimated_size`, which is the total `size()` of all frames added to the channel. A frame's `frame.size()` returns the total frame size plus the `FRAME_OVERHEAD`, which is currently 200 bytes per frame. However, only the frame data is written to the vec returned by `frame_data()`.

That said, the size of the returned will be overallocated by `200 * self.inputs.len()`. This may lead to OOM or at least unnecessary memory overgrowth.

Recommendation: Only account for the length of the frame data when calculating the final length of the resultant vec.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.25 Inefficient date type used for `BatchQueue` span cache

Severity: Low Risk

Context: batch_queue.rs#L55, batch_queue.rs#L82-L89

Description: The `BatchQueue` uses a `Vec<SingleBatch>` for the cache of spans. The `pop_next_batch()` function removes the first item from the cache via a `self.next_spans.remove(0)` call. This is an $\mathcal{O}(n)$ operation per batch, because removing the first item will result in shifting the rest of the vec values to the left. The total execution complexity for this operation would rise to $\mathcal{O}(n^2)$ for all the blocks in a span.

A malicious batcher could submit a valid, large span batch, which may result in consuming too much step/move time and hinder proof derivation.

Recommendation: Consider using a `VecDeque` with `pop_front()` instead.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.26 Deposits with mint above `u128::MAX` can halt derivation

Severity: Low Risk

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

Description: The `kona` derivation pipeline will emit a critical error if a deposit has a mint above `u128::MAX`.

crates/protocol/protocol/src/deposits.rs#L220-L224

```
let raw_mint: [u8; 16] = data[offset + 16..offset + 32].try_into().map_err(|_| {
    DepositError::MintDecode(Bytes::copy_from_slice(&data[offset + 16..offset + 32]))
})?;
tx.mint = u128::from_be_bytes(raw_mint);
```

For normal OP stack chains using ETH as the gas token this is **impossible** to trigger as you would need 3.403*10^20 ETH to trigger, but it could be relevant for chains using custom gas token with large decimals.

Recommendation: Allow mint value to be above `u128::MAX` this also requires changing the `op-alloy` types as it also enforces a mint of `u128`.

deposit.rs#L30:

```
/// The ETH value to mint on L2.
#[cfg_attr(feature = "serde", serde(default, with = "alloy_serde::quantity"))]
pub mint: u128,
```

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.27 Misalignment of L1 and L2 timestamps between data sources

Severity: Low Risk

Context: ethereum.rs#L67

Description: The Ethereum data source's `next()` function is called in the L1 retrieval stage of the pipeline, so the `block_ref` passed to it points to an L1 block whose timestamp may not be in sync with the L2's Ecotone activation requirements. The `cfg` already has a `blobs_enabled_l1_timestamp` value that is appears to be the value this check should use instead, though it is an `Option` so it may not always be available, and the value is not documented in the specs or used by the Go implementation.

Recommendation: Use a relevant L2 timestamp instead.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.28 Possible overflow in frame numbers allows malicious batchers to break frame queue ordering

Severity: Low Risk

Context: frame_queue.rs#L60-L106, frame.rs#L158

Description: The frame `number` of a `Frame` is a `u16` value. When `prune()` is called, the frame queue determines whether each two consecutive frames are in the same channel. When that is the case, the function determines if the two frames are contiguous but making sure `prev_frame.number` is exactly one bigger than `next_frame.number`, which will wrap around to zero if `prev_frame.number` has reached `u16:MAX`.

Recommendation: A malicious batcher could arrange frames such that the `next_frame` has number zero (while in the same channel as `prev_frame` to cause incorrect frame arrangements).

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.29 Incorrect handling of misaligned single batches

Severity: Low Risk

Context: single.rs#L35-L55

Description: The Holocene upgrade introduced the `Past` batch validity, where an outdated single batch whose timestamp is smaller than or equal to that of the L2 safe head is dropped without flushing the previous batch streaming and channel bank stages. Kona performs timestamp checks for single batches to determine their validity as follows:

```
pub fn check_batch_timestamp(
    &self,
    cfg: &RollupConfig,
    l2_safe_head: L2BlockInfo,
    inclusion_block: &BlockInfo,
) -> BatchValidity {
    let next_timestamp = l2_safe_head.block_info.timestamp + cfg.block_time;
    if self.timestamp > next_timestamp {
        if cfg.is_holocene_active(inclusion_block.timestamp) {
            return BatchValidity::Drop;
        }
        return BatchValidity::Future;
    }
    if self.timestamp < next_timestamp {
        if cfg.is_holocene_active(inclusion_block.timestamp) {
            return BatchValidity::Past;
        }
        return BatchValidity::Drop;
    }
```

```
    BatchValidity::Accept
}
```

While the spec for Holocene strictly states past batches are ones where the timestamp is exactly less than or equal to that of the L2 safe head, the logic in Kona incorrectly considers a batch whose timestamp is smaller than the next timestamp as past, without checking the case when a batch is misaligned (the case when `l2_safe_head.block_info.timestamp` < `self.timestamp` < `next_timestamp`). Such a case should instead return a `BatchValidity::Drop` result, triggering a flush for the whole span batch/channel.

Recommendation: Only return `BatchValidity::Past` if

```
self.timestamp <= l2_safe_head.block_info.timestamp
```

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.30 Batch parent hash check will always fail

Severity: Low Risk

Context: batch_validator.rs#L201-L276

Description: When `next_batch()` is called on the `BatchValidator` (used post Holocene), it sets the `parent_hash` value of the next batch to that of the `parent` block then calls `next_batch.check_batch()` passing the same block as the `parent` argument. As a result, the `if self.parent_hash != l2_safe_head.block_info.hash` statement in `check_batch()` will always return false and a batch with originally incorrect parent hash will pass this check, in case of a malicious batcher or a reorg. In comparison, the `BatchQueue` (used pre Holocene) only sets the batch's parent hash when `pop_next_batch()`.

Recommendation: Validate the parent hash before setting the value in the batch.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.31 Undecided span batches are silently lost

Severity: Low Risk

Context: batch_stream.rs#L187

Description: The `BatchStream` stage is used for buffering span batches post-Holocene. The function `next_batch()` is called by the next stage in the derivation pipeline to provide the next ready span batch. The function checks its buffer and, if it is empty, it calls back to the `ChannelReader` (the previous stage) to provide the next batch, which `take()`s the batch value while decompressing it so it is moved into the validity `match` block. If the span batch provided has an `Undecided` validity, then it is lost since the `BatchStream` does not store it anywhere before returning the `Err(PipelineError::NotEnoughData.temp())` error. The batch will never be processed later on when enough L1 information is available.

This results is incorrect derivation of payload attributes later on in the pipeline since the pipeline will be forced to derive empty batches in place of any dropped span batches.

Recommendation: Buffer the batch until enough information is available to determine its final validity.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.4.32 Incorrect precompile versions used post-Jovian

Severity: Low Risk

Context: provider.rs#L44-L64

Description: Jovian introduced input size restrictions for 4 precompiles. The current precompile version selection logic however returns:

1. The Isthmus version if the spec matches Interop or Jovian for precompiles.

2. The Isthmus version if the spec matches Interop.

Which is incorrect since the Interop upgrade comes after the Jovian upgrade. For accelerated precompiles, whenever spec ID 109 or 110 is activated, `accelerated_isthmus()` will be returned even if `accelerated_jovian()` was previously used for Jovian (spec ID 108). This will lead to different gas cost computations for the 4 highlighted precompiles, leading to state root divergence.

Recommendation: Return the correct precompiles for each of the upgrades.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 Misleading `batcher_address` variable name for `BlobSource`

Severity: Informational

Context: blobs.rs#L16-L33, blobs.rs#L40-L43

Description: The `batcher_address` variable name for `BlobSource` struct is misleading. This variable actually refers to the batch inbox address which is the recipient of the batcher transactions. However the variable name implies that this is the source of the transactions.

```
/// A data iterator that reads from a blob.
#[derive(Debug, Clone)]
pub struct BlobSource<F, B>
where
    F: ChainProvider + Send,
    B: BlobProvider + Send,
{
    /// Chain provider.
    pub chain_provider: F,
    /// Fetches blobs.
    pub blob_fetcher: B,
    /// The address of the batcher contract.
    pub batcher_address: Address,
    /// Data.
    pub data: Vec<BlobData>,
    /// Whether the source is open.
    pub open: bool,
}
```

Recommendation: Other than the misleading variable name, this variable is used correctly. However we recommend renaming it to `batch_inbox_address` to more accurately represent that this variable refers to the recipient of the batcher transactions and hence clear up any confusion.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.2 Possible panic with big prefix length

Severity: Informational

Context: node.rs#L165

Description: `path.slice()` will panic if `prefix.len()` is bigger than `path.length()` so a length check before the call is required.

Recommendation: Check that `prefix.len()` fits the buffer size.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.3 Consolidation can occur with insufficient transition states

Severity: Informational

Context: consolidation.rs#L61

Description: The documentation states that consolidation should occur when a `TransitionState` has a `Step` of 127, however that condition is not checked and consolidation can occur with any `PreState` variant.

Recommendation: A check making sure `boot_info.prestate` is an instance of `TransitionState` and that its `step` == `127` here would make sense.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.5.4 Incorrect warning output when no rollup config is found

Severity: Informational

Context: boot.rs#L133-L134

Description: The highlighted line will print all the values in `chain_ids` while it should only print the chain IDs that do not exist in `ROLLUP_CONFIGS`.

Recommendation: Only print the values not found in `ROLLUP_CONFIGS`.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.5.5 Documentation discrepancy in state transition

Severity: Informational

Context: pre_state.rs#L73-L77

Description: The returned `TransitionState` instance should have a `step` of `0` and an empty `pending_progress` vector, per the documentation.

Recommendation: This may be intentional but in case it is not, use the values from the documentation.

Coinbase: Acknowledged. We don't plan on supporting interop in the near future.

Cantina Managed: Acknowledged.

3.5.6 Handle types can be confused

Severity: Informational

Context: channel.rs#L30-L32

Description: The `FileDescriptor` type is generic for all input/output descriptors so `read_handler` can for example be `StdOut` or `HintWrite`.

Recommendation: Validation should be performed so `read_handler` can only be assigned a read-only descriptor and `write_handler` a write-only descriptor.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.7 Derivation can unexpectedly halt during a malformed system config update transaction

Severity: Informational

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

Description: During the derivation pipeline, the system config is updated from two locations:

1. During the `L1 Traversal` stage (for both polling and indexed):

crates/protocol/derive/src/stages/traversal/polling.rs#L103-L119:

```
match self.system_config.update_with_receipts(&receipts[..], addr, active) {
    Ok(true) => {
        let next = next_l1_origin.number as f64;
        kona_macros::set!(gauge, crate::Metrics::PIPELINE_LATEST_SYS_CONFIG_UPDATE,
        ↪  next);
        info!(target: "l1_traversal", "System config updated at block {next}.");
    }
    Ok(false) => { /* Ignore, no update applied */ }
    Err(err) => {
        error!(target: "l1_traversal", ?err, "Failed to update system config at block
        ↪  {}", next_l1_origin.number);
        kona_macros::set!(
            gauge,
            crate::Metrics::PIPELINE_SYS_CONFIG_UPDATE_ERROR,
            next_l1_origin.number as f64
        );
        return Err(PipelineError::SystemConfigUpdate(err).crit());
    }
}
```

2. During the `AttributesQueue` stage when preparing payload attributes:

crates/protocol/derive/src/attributes/stateful.rs#L101-L107:

```
    async fn prepare_payload_attributes(
        &mut self,
        l2_parent: L2BlockInfo,
        epoch: BlockNumHash,
    ) -> PipelineResult<OpPayloadAttributes> {
//...
            sys_config
                .update_with_receipts(
                    &receipts,
                    self.rollup_cfg.l1_system_config_address,
                    self.rollup_cfg.is_ecotone_active(header.timestamp),
                )
                .map_err(|e| PipelineError::SystemConfigUpdate(e).crit())?;
```

There are two issues with the implementation:

1. Firstly if `update_with_receipts` fails, the error returned is classified as `Critical` which causes the derivation pipeline to unexpectedly halt.
2. Secondly, the first error in the block will result in stopping the entire system update transaction for the entire block due to `try_for_each`.

crates/protocol/genesis/src/system/config.rs#L134-L145:

```
pub fn update_with_receipts(
    &mut self,
    receipts: &[Receipt],
    l1_system_config_address: Address,
    ecotone_active: bool,
) -> Result<bool, SystemConfigUpdateError> {
//...
        receipt.logs.iter().try_for_each(|log| {
            let topics = log.topics();
            if log.address == l1_system_config_address &&
                !topics.is_empty() &&
                topics[0] == CONFIG_UPDATE_TOPIC
            {
                // Safety: Error is bubbled up by the trailing `?`
                self.process_config_update_log(log, ecotone_active)?;
                updated = true;
            }
```

```
            Ok::<(), SystemConfigUpdateError>(())
        })?;
```

A scenario where this could occur is if a `setBatcherHash` transaction is made to the `SystemConfig` contract by the `SystemConfig` owner (accidentally or maliciously) with a `bytes32` value where any of the first 12 bytes of `_batcherHash` are non-zero.

SystemConfig.sol#L365-L369:

```
/// @notice Updates the batcher hash. Can only be called by the owner.
/// @param _batcherHash New batcher hash.
function setBatcherHash(bytes32 _batcherHash) external onlyOwner {
    _setBatcherHash(_batcherHash);
}
```

This emits a `ConfigUpdate(VERSION, UpdateType.BATCHER, data)` event where `data` is `bytes32`.

When ABI decoding the `ConfigUpdate` in Kona, during `process_config_update_log` the expected ABI of `data` is `address`. Hence, an `Err` will be returned when decoding the `ConfigUpdate` log.

crates/protocol/genesis/src/updates/batcher.rs#L46-L48:

```
let Ok(batcher_address) = <sol!(address)>::abi_decode_validate(validated.payload()) else
↪  {
    return Err(BatcherUpdateError::BatcherAddressDecodingError);
};
```

This will bubble up to `process_config_update_log`, which will result in both issues as mentioned above.

Recommendation: It is recommended to ignore any errors from malformed system config update transactions.

For reference, the `op-node` implementation ignores any error encountered during `UpdateSystemConfigWithL1Receipts`.

op-node/rollup/derive/l1_traversal.go#L78:

```
if err := UpdateSystemConfigWithL1Receipts(&l1t.sysCfg, receipts, l1t.cfg,
↪  nextL1Origin.Time); err != nil {
    // if UpdateSystemConfigWithL1Receipts returns an error, it is because one or more of
    ↪  the receipts are malformed or invalid
    // failure to apply is just informational, so we just log the error and continue
    l1t.log.Warn("failed to fully update L1 sysCfg with receipts from block", "block",
    ↪  nextL1Origin, "error", err)
    }
```

op-node/rollup/derive/attributes.go#L97-L99:

```
// errors from UpdateSystemConfigWithL1Receipts are ignored as they represent malformed
↪  or invalid updates
// and there is no recovery mechanism for malformed updates, we must process past them.
_ = UpdateSystemConfigWithL1Receipts(&sysConfig, receipts, ba.rollupCfg, info.Time())
```

And `UpdateSystemConfigWithL1Receipts` processes past any malformed system config update transactions, bundling all errors into a multierror.

op-node/rollup/derive/system_config.go#L42-L67:

```
// UpdateSystemConfigWithL1Receipts filters all L1 receipts to find config updates and
↪  applies the config updates to the given sysCfg
// Updates are applied individually, and any malformed or invalid updates are ignored.
// Any errors encountered during the update process are returned as a multierror.
func UpdateSystemConfigWithL1Receipts(sysCfg *eth.SystemConfig, receipts
↪  []*types.Receipt, cfg *rollup.Config, l1Time uint64) error {
    var result error
    for i, rec := range receipts {
        if rec.Status != types.ReceiptStatusSuccessful {
            continue
```

```
        }
        for j, log := range rec.Logs {
        // copy sysConfig to an update structure to preserve the original in case of
        ↪  error
            updated := *sysCfg
            if log.Address == cfg.L1SystemConfigAddress && len(log.Topics) > 0 &&
            ↪  log.Topics[0] ==       ConfigUpdateEventABIHash {
                err := ProcessSystemConfigUpdateLogEvent(&updated, log, cfg, l1Time)
                if err == nil {
                     // apply the updated structure
                    *sysCfg = updated
                } else {
                     // or append the error to the result
                     result = multierror.Append(result, fmt.Errorf("malformatted L1
                     ↪  system sysCfg log in receipt %d, log %d: %w", i, j, err))
                }
            }
        }
    }
    return result
}
```

The OP stack specifications are not specific on what the behaviour should be, so that should be fixed as well.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.8 Confusion on blob loading due to generic return value

Severity: Informational

Context: blobs.rs#L121-L123

Description: In the blobs data source, the `load_blobs()` function always returns `Ok()` as long as no errors arise during its execution. Since the returned value is the same whether the `BlobSource` is open or not, confusion for the calling logic may arise as no distinguishing value is returned for the case when the source is already open and no new blobs were loaded.

Recommendation: Add a value that signals if new blobs were loaded.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.9 DoS via OOM due to unlimited JSON payload loading

Severity: Informational

Context: chain_provider.rs#L271-L272

Description: JSON processing is expensive in general and the highlighted code snippet directly loads the JSON payload without length limits. An attacker can submit large calldata across a lot of transactions to drive up JSON processing, possibly leading to an OOM error and a DoS in block execution.

Recommendation: Limit the size of the JSON payload before it is loaded into the vector or use stream processing.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.10 Incorrect L1 origin used when re-orgs happen

Severity: Informational

Context: pipeline.rs#L141-L149

Description: The origin used in the reset signal is the same current L1 origin in the pipeline. That means that, once a re-org happens, the L1 origin is not reset/rewinded to a canonical origin anchored to the `l2_safe_head`, and is directly used in the reset signal.

Because `PollingTraversal` trusts the `l1_origin` in the reset signal and immediately overwrites its origin with it, this logic can lock the pipeline into a stale origin that never converges to the canonical chain after a reorg, or causes the pipeline to continue operating using a non-canonical origin context. In an FPVM/dispute context, this can make derivation non-terminating or yield an incorrect derived trace/output root, which can deterministically flip dispute outcomes.

Recommendation: Rewind the L1 origin to the latest canonical one when a re-org occurs.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.11 Incorrect update of `origin_index` result in inefficient code

Severity: Informational

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

Description: `SpanBatch::get_singular_batches` incorrectly updated `origin_index` to `origin_index = i` while searching `l1_origins[origin_index..]`. The code used `origin_index = i`, but `i` is relative to the sliced window, not the original `l1_origins`. This could cause subsequent lookups to restart from an earlier index in `l1_origins[origin_index..]` resulting in inefficient code.

```
/// Converts all [`SpanBatchElement`]s after the L2 safe head to [`SingleBatch`]es. The
/// resulting [`SingleBatch`]es do not contain a parent hash, as it is populated by the
/// Batch Queue stage.
pub fn get_singular_batches(
    &self,
    l1_origins: &[BlockInfo],
    l2_safe_head: L2BlockInfo,
) -> Result<Vec<SingleBatch>, SpanBatchError> {
    let mut single_batches = Vec::with_capacity(self.batches.len());
    let mut origin_index = 0;
    for batch in &self.batches {
//...
        let origin_epoch_hash = l1_origins[origin_index..l1_origins.len()]
            .iter()
            .enumerate()
            .find(|(_, origin)| origin.number == batch.epoch_num)
            .map(|(i, origin)| {
                origin_index = i; // bug
                origin.hash
            })
            .ok_or(SpanBatchError::MissingL1Origin)?;
//...
        single_batches.push(single_batch);
    }
    Ok(single_batches)
}
```

Recommendation:

```
      /// Converts all [`SpanBatchElement`]s after the L2 safe head to [`SingleBatch`]es.
       ↪ The
      /// resulting [`SingleBatch`]es do not contain a parent hash, as it is populated by
       ↪ the
      /// Batch Queue stage.
      pub fn get_singular_batches(
          &self,
          l1_origins: &[BlockInfo],
          l2_safe_head: L2BlockInfo,
      ) -> Result<Vec<SingleBatch>, SpanBatchError> {
```

```
          let mut single_batches = Vec::with_capacity(self.batches.len());
          let mut origin_index = 0;
          for batch in &self.batches {
  //...
              let origin_epoch_hash = l1_origins[origin_index..l1_origins.len()]
                  .iter()
                  .enumerate()
                  .find(|(_, origin)| origin.number == batch.epoch_num)
                  .map(|(i, origin)| {
-                     origin_index = i;
+                     origin_index += i;
                      origin.hash
                  })
                  .ok_or(SpanBatchError::MissingL1Origin)?;
  //...
              single_batches.push(single_batch);
          }
          Ok(single_batches)
      }
```

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.12 Incorrect key format for `PreimageKey::new_precompile`

Severity: Informational

Context: key.rs#L104-L116

Description: crates/proof/preimage/src/key.rs#L104-L116:

```
/// Creates a new precompile [PreimageKey] from a precompile address and input. The key
 ↪  will be
/// constructed as `keccak256(precompile_addr ++ input)`, and then the high-order byte of
 ↪  the
/// digest will be set to the type byte.
pub fn new_precompile(precompile_addr: [u8; 20], input: &[u8]) -> Self {
    let mut data = [0u8; 31];

    let mut hasher = Keccak256::new();
    hasher.update(precompile_addr);
    hasher.update(input);

    data.copy_from_slice(&hasher.finalize()[1..]);
    Self { data, key_type: PreimageKeyType::Precompile }
}
```

The correct preimage key format `keccak256(precompile_addr ++ gas ++ input)`, `PreimageKey::new_precompile` function is currently unused (instead callers use `new` directly, but can be a footgun in the future). Note the OP Stack specs are outdated. But we can see the on-chain `PreimageOracle` contract use this key format:

src/cannon/PreimageOracle.sol#L419-L426:

```
// copy precompile address, requiredGas, and input into memory to compute the key
mstore(ptr, shl(96, _precompile))
mstore(add(ptr, 20), shl(192, _requiredGas))
calldatacopy(add(28, ptr), _input.offset, _input.length)
// compute the hash
let h := keccak256(ptr, add(28, _input.length))
// mask out prefix byte, replace with type 6 byte
key := or(and(h, not(shl(248, 0xFF))), shl(248, 0x06))
```

Recommendation: Either remove this function as it is unused or correct the function to do `keccak256(precompile_addr ++ gas ++ input)`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.13 Incorrect post-Ecotone base fee scalar encoding in edge case

Severity: Informational

Context: variant.rs#L68-L81

Description: The following part of the spec is not implemented.

See the reference spec on System Config:

> **Post-Ecotone Scalar Encoding**
> - If there are non-zero bytes in the padding area, `baseFeeScalar` must be set to `MaxUint32`.

crates/protocol/protocol/src/info/variant.rs#L73-L86:

```
// --- Post-Ecotone Operations ---

let scalar = system_config.scalar.to_be_bytes::<32>();
let blob_base_fee_scalar = (scalar[0] == L1BlockInfoEcotone::L1_SCALAR)
    .then(|| {
        Ok::<u32, BlockInfoError>(u32::from_be_bytes(
            scalar[24..28].try_into().map_err(|_| BlockInfoError::L1BlobBaseFeeScalar)?,
        ))
    })
    .transpose()?
    .unwrap_or_default();
let base_fee_scalar = u32::from_be_bytes(
    scalar[28..32].try_into().map_err(|_| BlockInfoError::BaseFeeScalar)?,
);
```

There will only be non-zero bytes in the padding area in a misconfiguration so this is informative.

Recommendation: Implement the following:

> - If there are non-zero bytes in the padding area, `baseFeeScalar` must be set to `MaxUint32`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.14 Persistent block / blob `NotFound` errors are treated as temporary

Severity: Informational

Context: sources.rs#L49

Description: `NotFound` errors are treated as temporary errors - in the extremely unlikely scenario `kona` fails to obtain the block / blob from the backend because they are not found (for any reason) and the not found error is persistent, then the pipeline will be stuck instead of resetting to the safe head.

1. Blob not found returning `Backend` are treated as temporary.

crates/protocol/derive/src/errors/sources.rs#L38:

```
/// Error pertaining to the backend transport.
#[error("{0}")]
Backend(String),
```

2. Block not found returning `BlockNotFound` are treated as temporary.

crates/providers/providers-alloy/src/chain_provider.rs#L130-L132:

```
AlloyChainProviderError::BlockNotFound(id) => {
    Self::Temporary(PipelineError::Provider(format!("L1 Block not found: {id}")))
}
```

crates/providers/providers-alloy/src/l2_chain_provider.rs#L204-L206:

```
AlloyL2ChainProviderError::BlockNotFound(_) => {
    Self::Temporary(PipelineError::Provider("Block not found".to_string()))
}
```

Recommendation: See op-node reference for how it should be handled; change to reset error as they are persistent:

```
if err != nil {
    if errors.Is(err, ethereum.NotFound) {
        return nil, NewResetError(fmt.Errorf("failed to open blob data source: %w", err))
    }
    return nil, NewTemporaryError(fmt.Errorf("failed to open blob data source: %w", err))
}
```

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.15 Reorg check in indexed traversal throws `NextL1BlockHashMismatch` instead of `ReorgDetected`

Severity: Informational

Context: indexed.rs#L85

Description: In indexed traversal, the reorg check will throw `NextL1BlockHashMismatch` instead of `ReorgDetected` on a parent hash mismatch.

crates/protocol/derive/src/stages/traversal/indexed.rs#L83-L87:

```
if block.hash != block_info.parent_hash {
    return Err(
        ResetError::NextL1BlockHashMismatch(block.hash, block_info.parent_hash).reset()
    );
}
```

Callers of indexed traversal may use `ReorgDetected` errors to detect reorgs as opposed to `NextL1BlockHashMismatch` which mean they would not be able to detect reorgs effectively.

Recommendation: Consider using `ReorgDetected` instead.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.16 Future errors can be mapped to critical in `BatchStream` as it is possible to throw them post-Holocene

Severity: Informational

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

Description: `BatchStream` is only used post-Holocene (when `self.is_active()` is true), hence future (`BatchValidity::Future`) errors can be mapped to critical as it is impossible to emit this post-Holocene.

crates/protocol/derive/src/stages/batch/batch_stream.rs#L187-L189:

```
BatchValidity::Undecided | BatchValidity::Future => {
    return Err(PipelineError::NotEnoughData.temp());
}
```

Recommendation: `BatchValidity::Future` errors can be mapped to critical as it is impossible to emit this post-Holocene.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.17 `BatchQueue` L1 blocks store is pruned first before advancing origin

Severity: Informational

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

Description: The `BatchQueue` L1 blocks store is pruned first before advancing origin. This can lead to inefficiency because on the first batch which advances the L1 origin and stores the origin in `self.l1_blocks`, the `self.l1_blocks` will still contain past origins which are unneeded after origin advancement as `parent.l1_origin.number` will not match any origin number in `self.l1_blocks` since pruning occurs before origin advancement.

crates/protocol/derive/src/stages/batch/batch_queue.rs#L295-L339:

```
// If the epoch is advanced, update the l1 blocks.
// Advancing epoch must be done after the pipeline successfully applies the entire span
// batch to the chain.
// Because the span batch can be reverted during processing the batch, then we must
// preserve existing l1 blocks to verify the epochs of the next candidate batch.
if !self.l1_blocks.is_empty() && parent.l1_origin.number > self.l1_blocks[0].number {
    for (i, block) in self.l1_blocks.iter().enumerate() {
        if parent.l1_origin.number == block.number {
            self.l1_blocks.drain(0..i);
            info!(target: "batch_queue", "Advancing epoch");
            break;
        }
    }
    // If the origin of the parent block is not included, we must advance the origin.
}

// NOTE: The origin is used to determine if it's behind.
// It is the future origin that gets saved into the l1 blocks array.
// We always update the origin of this stage if it's not the same so
// after the update code runs, this is consistent.
let origin_behind =
    self.prev.origin().map_or(true, |origin| origin.number < parent.l1_origin.number);

// Advance the origin if needed.
// The entire pipeline has the same origin.
// Batches prior to the l1 origin of the l2 safe head are not accepted.
if self.origin != self.prev.origin() {
    self.origin = self.prev.origin();
    if origin_behind {
        // This is to handle the special case of startup.
        // At startup, the batch queue is reset and includes the
        // l1 origin. That is the only time where immediately after
        // reset is called, the origin behind is false.
        self.l1_blocks.clear();
    } else {
        let origin = match
        ↪  self.origin.as_ref().ok_or(PipelineError::MissingOrigin.crit()) {
            Ok(o) => o,
            Err(e) => {
                return Err(e);
            }
        };
        self.l1_blocks.push(*origin);
    }
    info!(target: "batch_queue", "Advancing batch queue origin: {:?}", self.origin);
}
```

Recommendation: Reorder:

```
// NOTE: The origin is used to determine if it's behind.
// It is the future origin that gets saved into the l1 blocks array.
// We always update the origin of this stage if it's not the same so
// after the update code runs, this is consistent.
let origin_behind =
    self.prev.origin().map_or(true, |origin| origin.number < parent.l1_origin.number);

// Advance the origin if needed.
// The entire pipeline has the same origin.
// Batches prior to the l1 origin of the l2 safe head are not accepted.
if self.origin != self.prev.origin() {
    self.origin = self.prev.origin();
    if origin_behind {
        // This is to handle the special case of startup.
        // At startup, the batch queue is reset and includes the
        // l1 origin. That is the only time where immediately after
        // reset is called, the origin behind is false.
        self.l1_blocks.clear();
    } else {
        let origin = match
        ↪  self.origin.as_ref().ok_or(PipelineError::MissingOrigin.crit()) {
            Ok(o) => o,
            Err(e) => {
                return Err(e);
            }
        };
        self.l1_blocks.push(*origin);
    }
    info!(target: "batch_queue", "Advancing batch queue origin: {:?}", self.origin);
}

// If the epoch is advanced, update the l1 blocks.
// Advancing epoch must be done after the pipeline successfully applies the entire span
// batch to the chain.
// Because the span batch can be reverted during processing the batch, then we must
// preserve existing l1 blocks to verify the epochs of the next candidate batch.
if !self.l1_blocks.is_empty() && parent.l1_origin.number > self.l1_blocks[0].number {
    for (i, block) in self.l1_blocks.iter().enumerate() {
        if parent.l1_origin.number == block.number {
            self.l1_blocks.drain(0..i);
            info!(target: "batch_queue", "Advancing epoch");
            break;
        }
    }
    // If the origin of the parent block is not included, we must advance the origin.
}
```

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.

3.5.18 `kona-executor` incorrectly sets pre EIP-161 `without_state_clear()` setting

Severity: Informational

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

Description: The `kona` executor incorrectly sets `revm` config `without_state_clear()` option to true.

crates/proof/executor/src/builder/core.rs#L254:

```
// Step 2. Create the executor, using the trie database.
let mut state = State::builder()
    .with_database(&mut self.trie_db)
    .with_bundle_update()
```

```
    .without_state_clear()
    .build();
```

`op-reth` does not have it set.

crates/payload/src/builder.rs#L339:

```
let mut db = State::builder().with_database(db).with_bundle_update().build();
```

The `without_state_clear()` option in `revm` is an option that enables pre EIP-161 (pre Spurious Dragon) behaviour where empty accounts that were touched within the transaction are persisted in state. Therefore it is a mistake to have the `without_state_clear()` option set post Spurious Dragon hardfork.

crates/database/src/states/cache.rs#L238-L246:

```
// Account is touched, but not selfdestructed or newly created.
// Account can be touched and not changed.
// And when empty account is touched it needs to be removed from database.
// EIP-161 state clear
if is_empty {
    if self.has_state_clear {
        // Touch empty account.
        this_account.touch_empty_eip161()
    } else {
        // If account is empty and state clear is not enabled we should save
        // empty account.
        this_account.touch_create_pre_eip161(changed_storage)
    }
}
```

The following is an integration test for `op-revm` that demonstrates how the `without_state_clear()` option is consensus-critical and can cause empty accounts to incorrectly persist in state (using a contract with 0 balance that `selfdestructs` to an empty account.).

This test can be placed in `crates/ee-tests/src` for the `bluealloy/revm` repository, or anywhere else that has access to the `revm` and `op_revm` packages.

```
use op_revm::{api::builder::OpBuilder, L1BlockInfo, OpContext, OpSpecId, OpTransaction};
use revm::{
    bytecode::opcode,
    context::TxEnv,
    database::{EmptyDB, State, BENCH_CALLER, BENCH_TARGET},
    primitives::{address, U256},
    state::{AccountInfo, Bytecode},
    ExecuteCommitEvm,
};

/// Standalone integration test: demonstrates that disabling EIP-161 state clear
↪  (`without_state_clear`)
/// while executing post-Spurious Dragon rules causes different persisted state.
///
/// This version runs through the OP execution pipeline (`op-revm`) at the Jovian
↪  hardfork.
#[test]
fn op_jovian_without_state_clear_diverges_on_touched_empty() {
    let beneficiary = address!("0x1000000000000000000000000000000000000001");

    // Runtime bytecode: PUSH20 <beneficiary> SELFDESTRUCT STOP
    let mut code_bytes = Vec::with_capacity(1 + 20 + 1 + 1);
    code_bytes.push(opcode::PUSH20);
    code_bytes.extend_from_slice(beneficiary.as_slice());
    code_bytes.push(opcode::SELFDESTRUCT);
    code_bytes.push(opcode::STOP);
    let runtime_code = Bytecode::new_legacy(code_bytes.into());

    let caller_info = AccountInfo::default()
        .with_balance(U256::from(1_000_000_000_000_000_000u128))
```

```
        .with_nonce(0);
    let contract_info = AccountInfo::default().with_code(runtime_code).with_nonce(1);

    // Provide L1 block info so OP fee logic doesn't try to fetch from the DB (EmptyDB).
    // Jovian implies Isthmus/Ecotone logic is active, so populate the option fields to
    ↪  avoid panics.
    let chain = L1BlockInfo {
        l2_block: Some(U256::ZERO),
        l1_base_fee: U256::ZERO,
        l1_fee_overhead: Some(U256::ZERO),
        l1_base_fee_scalar: U256::ZERO,
        l1_blob_base_fee: Some(U256::ZERO),
        l1_blob_base_fee_scalar: Some(U256::ZERO),
        operator_fee_scalar: Some(U256::ZERO),
        operator_fee_constant: Some(U256::ZERO),
        da_footprint_gas_scalar: Some(0),
        empty_ecotone_scalars: false,
        tx_l1_cost: None,
    };

    let run = |state_clear_enabled: bool| -> bool {
        let mut state = if state_clear_enabled {
            State::builder()
                .with_database(EmptyDB::default())
                .with_bundle_update()
                .build()
        } else {
            State::builder()
                .with_database(EmptyDB::default())
                .with_bundle_update()
                .without_state_clear()
                .build()
        };

        // Provide caller + contract in the prestate cache. Beneficiary is intentionally
        ↪  absent.
        state.insert_account(BENCH_CALLER, caller_info.clone());
        state.insert_account(BENCH_TARGET, contract_info.clone());

        let tx = OpTransaction::builder()
            .base(TxEnv::builder_for_bench())
            .build_fill();

        // Build an Optimism context explicitly (don't rely on `Context::mainnet()` which
        ↪  is a
        // trait-provided constructor for a different, mainnet-specific Context
        ↪  instantiation).
        let ctx = OpContext::new(state, OpSpecId::JOVIAN)
            .with_tx(tx.clone())
            .with_chain(chain.clone());

        let mut evm = ctx.build_op();

        let _ = evm.transact_commit(tx).unwrap();

        // Does the beneficiary show up as an explicit account entry?
        let mut beneficiary_in_trie = false;
        for (addr, _) in evm
            .0
            .ctx
            .journaled_state
            .database
            .cache
            .trie_account()
            .into_iter()
        {
```

```
            if addr == beneficiary {
                beneficiary_in_trie = true;
                break;
            }
        }
        beneficiary_in_trie
    };

    let with_clear = run(true);
    let without_clear = run(false);

    // Correct post-Spurious Dragon behavior: touched-empty must not persist.
    assert!(!with_clear);
    // With state clear disabled, the touched-empty account persists (diverges).
    assert!(without_clear);
}
```

> **Note:** On a discussion with OP Labs it was deemed that while the code looked out of place, a full kona program execution does not exhibit pre EIP-161 behavior. This is because `alloy-evm` ignores the `without_state_clear()` flag before block execution via `apply_pre_execution_changes()` in block.rs#L129.

Recommendation: Remove `without_state_clear()` in `kona-executor`.

Coinbase: Acknowledged.

Cantina Managed: Acknowledged.