BLOCKSEC

Security Audit
Report for Reth

Date: October 8, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About the Audit Target . . . 1
1.2 Disclaimer . . . 2
1.3 Procedure of Auditing . . . 2
1.3.1 Security Issues . . . 3
1.3.2 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **5**

2.1 Security Issue . . . 5
2.1.1 Inconsistent decoding of `_ethTxValue` across L2 clients . . . 5
2.1.2 Incorrect state handling of revived FIFO entries during deduplication . . . 6
2.1.3 Incorrect RPC extension hook composition prevents Mantle RPC initialization 7
2.1.4 Incorrect L1 data fee estimation in the function `mantle_arsia_check_funds()` 11
2.1.5 Double deduction of `SDM` refunds understates preconfirmation gas accounting . . . 13
2.1.6 Missing access and authorization lists from the L1 fee estimation envelope 15
2.2 Recommendation . . . 16
2.2.1 Include the L1 and operator fee in the balance pre-check of the function `handle_inner()` . . . 16
2.2.2 Perform transaction deduplication before preconfirmation admission checks 18
2.2.3 Prevent responder overwrites during concurrent preconfirmation retries for the same transaction hash . . . 20
2.3 Note . . . 22
2.3.1 Potential centralization risk . . . 22

Report Manifest

Item	Description
Client	Mantle
Target	Reth

Version History

Version	Date	Description
1.0	October 8, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-10-29 08:13:39+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Reth of Mantle.

Mantle is an Ethereum Layer 2 (L2) network built on the OP Stack, with MNT as its gas token and BVM-ETH as its L2 representation of ETH. This audit covers Mantle's customized fork of the Rust execution client reth, implemented in mantle-reth and op-reth. The client adds the Everest, Skadi, Limb, and Arsia hardforks and their activation rules, genesis construction for supported Mantle networks, and Layer 1 (L1) deposit processing with MNT and BVM-ETH minting and transfers. It also implements a frozen base fee mechanism and a data availability (DA) fee model with L1 data fees, operator fees, and a token ratio. Its preconfirmation transaction flow uses a dedicated transaction pool, a remote procedure call (RPC) entry point, and a journal for recovery after a restart. Additional RPC methods provide Mantle-specific fee estimation.

Note this audit focuses on the codes located in the following directories/files:

- op-reth/crates/consensus/src/lib.rs
- op-reth/crates/evm/src/l1.rs
- op-reth/crates/evm/src/build.rs
- op-reth/crates/node/src/engine.rs
- op-reth/crates/payload/src/validator.rs
- op-reth/crates/rpc/src/eth/receipt.rs
- op-reth/crates/rpc/src/eth/call.rs
- op-reth/crates/rpc/src/eth/transaction.rs
- op-reth/crates/rpc/src/engine.rs
- op-reth/crates/rpc/src/error.rs
- op-reth/crates/rpc/src/sequencer.rs
- op-reth/crates/txpool/src/validator.rs
- mantle-reth/crates/chainspec/
- mantle-reth/crates/cli/
- mantle-reth/crates/eth-api/
- mantle-reth/crates/rpc-ext/
- mantle-reth/crates/preconf/
- patches/reth-provider/src/providers/state/mod.rs
- patches/reth-provider/src/providers/state/latest.rs

[1] https://github.com/mantle-xyz/reth

- patches/reth-trie-common/src/proofs.rs(StorageMultiProof/DecodedStorageMultiProof::storage_pr
- patches/reth-transaction-pool/src/traits.rs/extra_balance_cost()
- patches/reth-transaction-pool/src/pool/txpool.rs
- patches/reth-transaction-pool/src/lib.rs
- patches/reth-transaction-pool/src/validate/eth.rs
- patches/reth-transaction-pool/src/validate/mod.rs

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
Reth	`Version 1`	`c4a1f11803d2bc37cd6e8a4e3bb8737c0ca23705`
	`Version 2`	`bcc99c42909291948a79e0c84467e2cc20e8e95c`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **six** potential security issues. Besides, we have **three** recommendations and **one** note.

- Medium Risk: 2
- Low Risk: 4
- Recommendation: 3
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Inconsistent decoding of `_ethTxValue` across L2 clients	Security Issue	Fixed
2	Medium	Incorrect state handling of revived FIFO entries during deduplication	Security Issue	Fixed
3	Low	Incorrect RPC extension hook composition prevents Mantle RPC initialization	Security Issue	Confirmed
4	Low	Incorrect L1 data fee estimation in the function `mantle_arsia_check_funds()`	Security Issue	Fixed
5	Low	Double deduction of `SDM` refunds understates preconfirmation gas accounting	Security Issue	Confirmed
6	Low	Missing access and authorization lists from the L1 fee estimation envelope	Security Issue	Fixed
7	-	Include the L1 and operator fee in the balance pre-check of the function `handle_inner()`	Recommendation	Confirmed
8	-	Perform transaction deduplication before preconfirmation admission checks	Recommendation	Fixed
9	-	Responder overwrite during concurrent preconfirmation retries for same transaction hash	Recommendation	Fixed
10	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Inconsistent decoding of `_ethTxValue` across L2 clients

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the L1 bridge, the function `depositTransaction()` records the user-supplied `_ethTxValue` in the `TransactionDeposited` event as a `uint256`. The Go client, `op-geth`, decodes the complete 256-bit value in the function `unmarshalDepositVersion1()` and attempts to transfer the corresponding amount of BVM-ETH to the recipient. However, the Rust client truncates the value to a `u128` during deposit derivation, discarding the upper 128 bits. For example, an

`_ethTxValue` of 2^{128} is interpreted as 2^{128} by `op-geth` but as zero by the Rust client. A user does not need to hold the specified amount to trigger this discrepancy because the transaction derived by op-geth reverts due to insufficient balance, whereas the transaction derived by the Rust client succeeds with a zero-value transfer. The two clients therefore produce different execution results for the same L1 deposit event.

Impact An attacker can submit a deposit that produces inconsistent state transitions across the two clients, causing the L2 network to split.

Suggestion Enforce a `u128` upper bound on `_ethTxValue` in the function `depositTransaction()` and ensure that both clients reject values exceeding this limit during deposit derivation.

2.1.2 Incorrect state handling of revived FIFO entries during deduplication

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `dispatch.rs`, when a transaction is rejected during a payload build, its hash is recorded as excluded in `loop_state`, and the corresponding FIFO entry is marked as terminal. If the same transaction hash is resubmitted within the same build cycle, the FIFO entry is revived and transitioned back to the `Waiting` state. However, the deduplication check still retrieves the previous exclusion reason and invokes the function `cancel_responder()`, which returns the previous error to the new responder without updating the FIFO entry's status. Consequently, the entry remains in the `Waiting` state and may be selected during a subsequent payload build and included on-chain, even though the resubmitted transaction was reported as rejected.

```
257    if let Some(reason) = loop_state.excluded_reason(&hash).cloned() {
258        if matches!(reason, PreconfError::Timeout { .. }) {
259            trace!(
260                target: "mantle::preconf::dispatch",
261                ?hash,
262                "prior exclusion was Timeout; clearing to re-evaluate against refreshed inserted_at
                    "
263            );
264            loop_state.clear_excluded(&hash);
265            // Fall through to gate evaluation.
266        } else {
267            trace!(
268                target: "mantle::preconf::dispatch",
269                ?hash, ?reason,
270                "dedup hit; forwarding prior rejection to any pending responder"
271            );
272            fifo.cancel_responder(&hash, reason).await;
```

Listing 2.1: mantle-reth/crates/preconf/src/builder/dispatch.rs

```
742    pub async fn cancel_responder(&self, hash: &TxHash, err: PreconfError) {
743        let responder = {
744            let mut inner = self.inner.lock().await;
```

```
745          let r = inner
746              .entries
747              .get_mut(hash)
748              .and_then(|e| e.responder.take())
749              .or_else(|| inner.pending_responders.remove(hash).map(|(_, r)| r));
750          // Drop any ghost pending responder (invariant #2 violation);
751          // no-op in the normal case.
752          inner.pending_responders.remove(hash);
```

Listing 2.2: mantle-reth/crates/preconf/src/preconf_tx_set.rs

Impact A resubmitted transaction may be reported as rejected but subsequently included in the next block, resulting in behavior that contradicts the builder's response.

Suggestion Remove the stale rejection forwarding logic for revived entries and keep the responder pending until a subsequent payload build determines the transaction's actual execution result.

2.1.3 Incorrect RPC extension hook composition prevents Mantle RPC initialization

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the function `launch_with_proof_history()` in the file `main.rs`, the function `node()` configures the builder with the add-ons returned by the function `MantleNode::add_ons()`. As part of this configuration, the function `extend_rpc_modules()` registers a hook that installs the Mantle-specific RPC methods and initializes the preconfirmation canonical-state handler and journal-maintenance task. The same builder is then passed to the function `with_proofs_history_launch_ctx()`, where the macro `install_proofs_history!` invokes the function `extend_rpc_modules()` again to register the proof-history RPC methods. However, the second call replaces the previously registered hook instead of composing with it. Consequently, when proof history is enabled, only the proof-history hook is executed during node startup, while the Mantle RPC extensions and the related preconfirmation maintenance components are not initialized.

```
139  async fn launch_with_proof_history<S>(
140      builder: WithLaunchContext<NodeBuilder<DatabaseEnv, OpChainSpec>>,
141      node: MantleNode,
142      args: RollupArgs,
143      mdbx: Arc<S>,
144      preconf_enabled: bool,
145  ) -> eyre::Result<(), ErrReport>
146  where
147      S: OpProofsStore + DatabaseMetrics + Send + Sync + 'static,
148  {
149      let metrics_mdbx = mdbx.clone();
150
151      let builder = builder.node(node).on_node_started(move |full_node| {
```

```
152         seed_blockchain_tree_metrics(&full_node.provider);
153         // Pre-register preconf metric series (0 baseline). Preconf-only.
154         if preconf_enabled {
155             seed_preconf_metrics();
156         }
157         spawn_proofs_db_metrics(
158             full_node.task_executor,
159             metrics_mdbx,
160             full_node.config.metrics.push_gateway_interval,
161         );
162         Ok(())
163     });
164
165     let handle = with_proofs_history_launch_ctx(builder, &args, mdbx).launch().await?;
166
167     handle.node_exit_future.await
168 }
```

Listing 2.3: mantle-reth/crates/cli/src/main.rs

```
470                         .task_executor()
471                         .spawn_critical_task("mantle-preconf-canon-handler", canon.run());
472                     info!(target: "reth::cli", "Mantle preconf canonical-state handler spawned")
                            ;
473
474                     // If persistence is on, drive the periodic rotation
475                     // loop under the reth `TaskManager`'s graceful-
476                     // shutdown protocol. Holding the `GracefulShutdownGuard`
477                     // across `run_rejournal_loop`'s final rotate makes the
478                     // `TaskManager` wait for the last on-disk write to
479                     // finish before returning from process shutdown, so
480                     // sealed hashes reported by the canon handler on the
481                     // way down are dropped from the journal file before
482                     // the node exits.
483                     let journal = svc.journal().clone();
484                     let interval = svc.cfg().rejournal_interval;
485                     ctx.node().task_executor().spawn_critical_with_graceful_shutdown_signal(
486                         "mantle-preconf-rejournal-loop",
487                         move |signal| async move {
488                             // `signal` (`GracefulShutdown`) resolves to
489                             // a `GracefulShutdownGuard` when the reth
490                             // `TaskManager` begins shutdown. Passing it
491                             // as the shutdown future to
492                             // `run_rejournal_loop` -whose `T` type
493                             // parameter carries the guard through the
494                             // final-rotate step -keeps the guard alive
495                             // until the last on-disk write finishes,
496                             // so the process only exits after the
497                             // journal file has been closed cleanly.
498                             let guard =
499                                 mantle_reth_preconf::run_rejournal_loop(journal, interval, signal
                                        )
500                                     .await;
```

```
501                             // Explicit drop for clarity; the guard is
502                             // released here, letting `TaskManager`'s
503                             // outstanding-tasks counter reach zero.
504                             drop(guard);
505                         },
506                     );
507                     info!(target: "reth::cli", "Mantle preconf journal rotation loop spawned");
508
509                     let handler =
510                         svc.rpc_handler(ctx.node().pool().clone(), ctx.node().provider().clone())
                                ;
511                     Arc::new(handler) as Arc<dyn mantle_reth_rpc_ext::DynPreconfHandler>
512                 });
513
514             let mantle_ext = MantleRpcExt::new(
515                 ctx.node().provider().clone(),
516                 Arc::new(ctx.registry.eth_api().clone()),
517                 sequencer_client,
518                 preconf_handler,
519             );
520             // `replace_configured`, not `merge_configured`: this module now also serves
521             // `eth_simulateV1`, which the standard `eth_` namespace already registers. Merging
                    a
522             // duplicate method name fails, so remove-then-add to override it. The other
                    methods in
523             // this module are new names and are unaffected by the choice.
524             ctx.modules.replace_configured(mantle_ext.into_rpc())?;
525             info!(target: "reth::cli", "Mantle RPC extensions registered");
526
527             Ok(())
528         });
529         add_ons
530     }
531 }
532
533 #[cfg(test)]
534 mod tests {
535     use super::*;
536     use mantle_reth_preconf::PreconfConfig;
537     use reth_optimism_node::args::RollupArgs;
538
539     fn default_args() -> RollupArgs {
540         RollupArgs::default()
541     }
542
543     #[test]
544     fn new_starts_with_preconf_disabled() {
545         // Default constructor: preconf opt-in, must be None at entry. Regression
546         // guard for the "MantleNode behaves exactly like OpNode when preconf is
547         // not configured" contract.
548         let node = MantleNode::new(default_args());
549         assert!(node.preconf.is_none());
550     }
```

```
551
552     /// Build a service builder with a fresh journal under a temp dir.
553     /// The journal is mandatory, so construction opens a real file.
554     async fn test_svc(dir: &tempfile::TempDir) -> PreconfServiceBuilder {
555         let cfg = PreconfConfig {
556             journal_path: Some(dir.path().join("preconf.jsonl")),
557             ..PreconfConfig::default()
558         };
559         PreconfServiceBuilder::from_config(cfg).await.expect("default validates + journal opens
                ")
560     }
561
562     #[tokio::test]
563     async fn with_preconf_attaches_service_builder() {
564         // `with_preconf` must store the service builder reachable through
```

Listing 2.4: mantle-reth/crates/cli/src/node.rs

```
44  macro_rules! install_proofs_history {
45      ($builder:expr, $args:expr, $mdbx:expr) => {{
46          let storage: OpProofsStorage<Arc<_>> = $mdbx.into();
47          let storage_exec = storage.clone();
48
49          let RollupArgs {
50              proofs_history_window,
51              proofs_history_prune_interval,
52              proofs_history_verification_interval,
53              ..
54          } = *$args;
55
56          $builder
57              .install_exex("proofs-history", async move |exex_context| {
58                  Ok(OpProofsExEx::builder(exex_context, storage_exec)
59                      .with_proofs_history_window(proofs_history_window)
60                      .with_proofs_history_prune_interval(proofs_history_prune_interval)
61                      .with_verification_interval(proofs_history_verification_interval)
62                      .build()
63                      .run()
64                      .boxed())
65              })
66              .extend_rpc_modules(move |ctx| {
67                  info!(
68                      target: "reth::cli",
69                      "Installing proofs-history RPC overrides (eth_getProof, debug_*)"
70                  );
71                  let api_ext = EthApiExt::new(ctx.registry.eth_api().clone(), storage.clone());
72                  // `Attrs` is a phantom parameter on `DebugApiExt` -nothing in `new()` pins it,
                        so
73                  // in a generic context it has to be named explicitly for `into_rpc()` to
                        resolve.
74                  let debug_ext: DebugApiExt<
75                      _,
76                      _,
```

```
77                      _,
78                      _,
79                      OpPayloadBuilderAttributes<OpTransactionSigned>,
80                  > = DebugApiExt::new(
81                      ctx.node().provider().clone(),
82                      ctx.registry.eth_api().clone(),
83                      storage,
84                      ctx.node().task_executor().clone(),
85                      ctx.node().evm_config().clone(),
86                  );
87                  ctx.modules.replace_configured(api_ext.into_rpc())?;
88                  ctx.modules.replace_configured(debug_ext.into_rpc())?;
89                  info!(target: "reth::cli", "Proofs-history RPC overrides installed");
90                  Ok(())
91              })
92      }};
93  }
```

Listing 2.5: mantle-reth/crates/cli/src/proofs_history.rs

Impact The Mantle RPC extension hook is not executed, which prevents the Mantle RPC functionality and related preconfirmation maintenance from being initialized.

Suggestion Compose the Mantle and proof-history RPC registration logic so that both hooks are preserved and executed during node startup.

Feedback from the project Confirmed by the project team and deferred to a future release. The team clarified that the affected configuration is not used for preconfirmation in the current deployment, limiting the issue's practical impact, and that a fix is planned for the next release.

2.1.4 Incorrect L1 data fee estimation in the function `mantle_arsia_check_funds()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `call.rs`, the function `eth_estimateGas()` invokes the function `mantle_arsia_check_funds()` to determine whether the sender's balance can cover the transaction value, L2 execution fee, L1 data fee, and operator fee. The function estimates the L1 data fee from the transaction input with an additional fixed length of 80 bytes. In contrast, `op-geth` constructs a complete dynamic-fee transaction and calculates the L1 data fee from the entire encoded transaction. For a native asset transfer with empty input, the current implementation calculates an L1 data fee of zero, although the complete signed transaction submitted to the transaction pool still incurs an L1 data fee. Consequently, the function `eth_estimateGas()` may consider the sender's balance sufficient, while the actual transaction is rejected during transaction pool validation as the balance cannot cover the complete fee.

```
69  let l1_cost =
70      check.l1_block_info.calculate_tx_l1_cost_for_estimate(check.tx_input, spec_id, 80);
```

Listing 2.6: mantle-reth/crates/eth-api/src/lib.rs

```
100             EstimateCall::estimate_gas_at(self, request.clone(), at, state_override).await?;
101
102         // [MANTLE] Post-estimation Arsia balance check (op-geth v1.5.5 mantleArsiaCheckFunds)
103         // geth uses target block state (opts.State from StateAndHeaderByNumberOrHash).
104         let chain_spec = self.provider().chain_spec();
105         if chain_spec.is_mantle() &&
106             let Ok(Some(block)) = self.provider().block_by_id(at) &&
107             chain_spec.is_mantle_arsia_active_at_timestamp(block.header().timestamp())
108         {
109             let fee_cap = U256::from(
110                 request
111                     .as_ref()
112                     .max_fee_per_gas
113                     .unwrap_or(request.as_ref().gas_price.unwrap_or(0)),
114             );
115             if !fee_cap.is_zero() &&
116                 let Ok(mut l1_block_info) = extract_l1_info(block.body()) &&
117                 let Ok(state) = self.provider().state_by_block_id(at) &&
118                 let Some(from) = request.as_ref().from
119             {
120                 if let Ok(Some(ratio)) = state.storage(
121                     op_revm::constants::GAS_ORACLE_CONTRACT,
122                     op_revm::constants::TOKEN_RATIO_SLOT.into(),
123                 ) {
124                     l1_block_info.token_ratio = ratio;
125                 }
126                 let balance = from_override_balance
127                     .or_else(|| state.account_balance(&from).ok().flatten())
128                     .unwrap_or(U256::ZERO);
129                 let input = request.as_ref().input.input().cloned().unwrap_or_default();
130
131                 if let Err(e) = mantle_reth_eth_api::mantle_arsia_check_funds(
132                     &mantle_reth_eth_api::ArsiaFundsCheck {
133                         gas_limit: estimate.try_into().unwrap_or(u64::MAX),
134                         fee_cap,
135                         value: request.as_ref().value.unwrap_or(U256::ZERO),
136                         from_balance: balance,
137                         l1_block_info: &l1_block_info,
138                         tx_input: &input,
139                         chain_spec: chain_spec.as_ref(),
140                         timestamp: block.header().timestamp(),
141                     },
142                 ) {
143                     return Err(
144                         reth_rpc_eth_types::EthApiError::InvalidParams(e.to_string()).into()
145                     );
146                 }
147             }
148         }
149
150         Ok(estimate)
```

Listing 2.7: op-reth/crates/rpc/src/eth/call.rs

Impact A sender may receive a successful gas estimate but subsequently have the transaction rejected because the actual L1 data fee was not included in the balance check.

Suggestion Calculate the L1 data fee from a complete encoded transaction consistent with transaction pool validation, including transactions with empty input.

2.1.5 Double deduction of SDM refunds understates preconfirmation gas accounting

Severity Low

Status Confirmed

Introduced by Version 1

Description In file apply.rs, the function apply_preconf_tx() executes a preconfirmation transaction and invokes the function build_receipt() with the resulting ExecutionResult to construct an internal PreconfReceipt. The gas_used field of this receipt is set to the value returned by the function tx_gas_used() and is subsequently used to update the shared block gas usage, the preconfirmation gas budget, and the payload value.

When the gas floor is not applied, the upstream function canonicalize_result_gas() has already deducted the Sequencer-Defined Metering (SDM) refund from the transaction's gas consumption. However, the function tx_gas_used() deducts the regular refund counter, which includes the same SDM refund, from the canonicalized gas value again. Consequently, the SDM refund is deducted twice, causing the internal receipt and the related preconfirmation accounting to record less gas than the canonical execution result.

```
65  pub fn apply_preconf_tx<B>(
66      builder: &mut B,
67      tx: impl ExecutorTx<B::Executor>,
68      tx_hash: TxHash,
69      block_height: u64,
70  ) -> Result<PreconfReceipt, ApplyError>
71  where
72      B: BlockBuilder,
73  {
74      let mut captured: Option<PreconfReceipt> = None;
75      let exec_result = builder.execute_transaction_with_result_closure(tx, |res| {
76          let ras = res.result();
77          captured = Some(build_receipt(tx_hash, block_height, &ras.result));
78      });
79      // Route the raw `BlockExecutionError` (variant intact) into the
80      // classifier -stringifying here would erase the InvalidTx-vs-fatal
81      // distinction the caller needs.
82      interpret_apply_result(exec_result.map(|_| ()), captured)
83  }
```

Listing 2.8: mantle-reth/crates/preconf/src/apply.rs

```rust
pub fn build_receipt<H: core::fmt::Debug + 'static>(
    tx_hash: TxHash,
    block_height: u64,
    result: &ExecutionResult<H>,
) -> PreconfReceipt {
    match result {
        ExecutionResult::Success { gas, logs, .. } => PreconfReceipt {
            tx_hash,
            block_height,
            status: true,
            logs: logs.clone(),
            gas_used: gas.tx_gas_used(),
            reason: String::new(),
            revert_data: Bytes::new(),
        },
        ExecutionResult::Revert { gas, output, .. } => {
            // Decode `Error(string)` →"execution reverted: <msg>", matching
            // op-geth's `abi.UnpackRevert`. Panic/custom/raw reverts miss the
            // selector →`Err` →bare "execution reverted".
            let reason = match Revert::abi_decode(output.as_ref()) {
                Ok(r) => format!("execution reverted: {}", r.reason),
                Err(_) => "execution reverted".to_string(),
            };
            PreconfReceipt {
                tx_hash,
                block_height,
                status: false,
                // revm surfaces "logs emitted before revert" as an
                // observability field, but the EVM's state rollback erases
                // them from statedb. Op-geth reads receipts from statedb
                // (`state_processor.go:281`) so its Failed-status receipts
                // observe empty logs naturally. To keep our preconf receipt
                // byte-equal with the eventually-sealed receipt, we drop
                // revm's pre-revert log snapshot here.
                logs: Vec::new(),
                gas_used: gas.tx_gas_used(),
                reason,
                revert_data: output.clone(),
            }
        }
        ExecutionResult::Halt { reason, gas, .. } => {
            // Map to op-geth's vm-error text when the concrete halt type is
            // known (production: OpHaltReason). build_receipt is generic over
            // the EVM's abstract HaltReason, so we downcast rather than thread
            // a new trait bound through the (deliberately generic) builder
            // path; unknown halt types keep their opaque Debug form.
            let any = reason as &dyn Any;
            let reason = any
                .downcast_ref::<OpHaltReason>()
                .map(|h| h.geth_halt_reason())
                .or_else(|| any.downcast_ref::<HaltReason>().map(|h| h.geth_halt_reason()))
                .unwrap_or_else(|| format!("{reason:?}"));
```

```
181             PreconfReceipt {
182                 tx_hash,
183                 block_height,
184                 status: false,
185                 logs: Vec::new(),
186                 gas_used: gas.tx_gas_used(),
187                 reason,
188                 revert_data: Bytes::new(),
189             }
190         }
191     }
192 }
```

Listing 2.9: mantle-reth/crates/preconf/src/apply.rs

Impact The understated gas usage may allow subsequent transactions to pass the preconfirmation gas admission checks despite insufficient remaining block gas, causing them to fail the executor's authoritative remaining block gas check. The payload value may also be understated.

Suggestion Use the canonical gas-used value from the post-execution `GasOutput` for the internal receipt and all related gas and fee calculations without deducting the `SDM` refund again.

Feedback from the project Confirmed by the project team and deferred to a future release. The team clarified that the affected code path is unreachable under Mantle's current configuration and that the upstream fix will be incorporated in the next upstream sync.

2.1.6 Missing access and authorization lists from the L1 fee estimation envelope

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `build_unsigned_tx_envelope()`, the transaction envelope used to estimate the L1 data fee is reconstructed as either an EIP-1559 transaction or a legacy transaction. When constructing an EIP-1559 transaction, the function always initializes the access list as empty instead of preserving the access list provided in the request. The function also does not construct an EIP-7702 transaction and therefore omits the request's authorization list entirely. In contrast, `op-geth` preserves these fields when constructing the transaction used for fee estimation. Consequently, a request containing an access list or EIP-7702 authorizations is encoded without all of its data, causing the calculated L1 data fee to be lower than the fee charged for the actual transaction.

```
764 fn build_unsigned_tx_envelope(
765     request: &TransactionRequest,
766     gas_estimate: U256,
767     base_fee: u64,
768     chain_id: u64,
769 ) -> Vec<u8> {
770     use alloy_consensus::{SignableTransaction, TxEip1559, TxLegacy};
771     use alloy_eips::eip2718::Encodable2718;
```

```
772         use alloy_primitives::Signature;
773
774         let gas_limit: u64 = gas_estimate.try_into().unwrap_or(u64::MAX);
775         let to = request.to.unwrap_or(TxKind::Create);
776         let value = request.value.unwrap_or(U256::ZERO);
777         let input = request.input.input().cloned().unwrap_or_default();
778         let nonce = request.nonce.unwrap_or(0);
779         let zero_sig = Signature::new(U256::ZERO, U256::ZERO, false);
780
781         if base_fee > 0 && request.gas_price.is_none() {
782             TxEip1559 {
783                 chain_id,
784                 nonce,
785                 max_fee_per_gas: request.max_fee_per_gas.unwrap_or(0),
786                 max_priority_fee_per_gas: request.max_priority_fee_per_gas.unwrap_or(0),
787                 gas_limit,
788                 to,
789                 value,
790                 input,
791                 access_list: Default::default(),
792             }
793             .into_signed(zero_sig)
794             .encoded_2718()
795         } else {
796             TxLegacy {
797                 chain_id: None,
798                 nonce,
799                 gas_price: request.gas_price.unwrap_or(0),
800                 gas_limit,
801                 to,
802                 value,
803                 input,
804             }
805             .into_signed(zero_sig)
806             .encoded_2718()
807         }
808     }
```

Listing 2.10: mantle-reth/crates/rpc-ext/src/lib.rs

Impact Transactions containing an access list or EIP-7702 authorizations may receive an L1 data fee estimate that is lower than their actual L1 cost.

Suggestion Preserve the transaction type, access list, and authorization list when constructing the envelope used for L1 data fee estimation.

2.2 Recommendation

2.2.1 Include the L1 and operator fee in the balance pre-check of the function `handle_inner()`

Status Confirmed

Introduced by Version 1

Description In the function `handle_inner()`, the synchronous balance pre-check calculates the required balance using the transaction cost and the value returned by the function `extra_balance_cost()`. However, the transaction is constructed through the function `from_pooled()`, which initializes `extra_balance_cost` to zero. The validator does not calculate and store the additional balance cost representing the L1 data fee and operator fee until the function `pool.add()`. Consequently, a transaction whose cumulative cost exceeds the sender's balance may pass the initial pre-check but subsequently remain non-executable in the transaction pool due to insufficient funds. The RPC caller must then wait until the preconfirmation request times out instead of receiving an immediate insufficient funds error.

```
158       if nonce == pending_nonce {
159           let own_cost = pool_tx.cost().saturating_add(pool_tx.extra_balance_cost());
160           let required = committed_cost.saturating_add(own_cost);
161           if own_cost <= on_chain_balance && required > on_chain_balance {
162               debug!(
163                   target: "mantle::preconf::rpc",
164                   ?sender, ?nonce, %required, %on_chain_balance,
165                   "insufficient funds rejected"
166               );
167               return Err(preconf_error_to_rpc(&PreconfError::InsufficientFunds {
168                   balance: on_chain_balance,
169                   required,
170               }));
171           }
172       }
```

Listing 2.11: mantle-reth/crates/preconf/src/rpc.rs

```
64    pub fn new(transaction: Recovered<Cons>, encoded_length: usize) -> Self {
65        Self {
66            inner: EthPooledTransaction::new(transaction, encoded_length),
67            estimated_tx_compressed_size: Default::default(),
68            conditional: None,
69            interop: Arc::new(AtomicU64::new(NO_INTEROP_TX)),
70            _pd: core::marker::PhantomData,
71            encoded_2718: Default::default(),
72            extra_balance_cost: U256::ZERO,
73        }
74    }
```

Listing 2.12: op-reth/crates/txpool/src/transaction.rs

```
346           // Checks for max cost
347           if cost > balance {
348               return TransactionValidationOutcome::Invalid(
349                   valid_tx.into_transaction(),
350                   InvalidTransactionError::InsufficientFunds(
351                       GotExpected { got: balance, expected: cost }.into(),
352                   )
353                   .into(),
354               );
```

```
355         }
356
357         // [MANTLE] Stash the overlay so the pool's cumulative accounting reserves it too. OP
358         // txs are never EIP-4844, so the sidecar-less 'Valid' variant is correct.
359         let mut tx = valid_tx.into_transaction();
360         tx.set_extra_balance_cost(cost_addition);
```

Listing 2.13: op-reth/crates/txpool/src/validator.rs

Suggestion Calculate and store the L1 data fee and operator fee in `extra_balance_cost` before performing the synchronous balance pre-check in the function `handle_inner()`, using the same calculation as the transaction pool validator.

Feedback from the project Confirmed by the project team, remediation is planned for the next release.

2.2.2 Perform transaction deduplication before preconfirmation admission checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `payload_builder.rs`, the function `admit_and_dispatch()` performs the same-sender cascade check and block-capacity admission before invoking the function `apply_one_preconf()`, where the transaction deduplication check is performed. Consequently, a transaction that has already been committed, excluded, or otherwise processed may undergo admission handling before its previous status is identified. If the transaction is deferred or rejected during admission, the deduplication check is never reached, and the transaction may be handled as a new candidate rather than according to its existing result.

```
253   if loop_state.is_committed(&hash) {
254       trace!(target: "mantle::preconf::dispatch", ?hash, "dedup hit; already committed");
255       return Ok(());
256   }
257   if let Some(reason) = loop_state.excluded_reason(&hash).cloned() {
258       if matches!(reason, PreconfError::Timeout { .. }) {
259           trace!(
260               target: "mantle::preconf::dispatch",
261               ?hash,
262               "prior exclusion was Timeout; clearing to re-evaluate against refreshed inserted_at
                    "
263           );
264           loop_state.clear_excluded(&hash);
265           // Fall through to gate evaluation.
266       } else {
267           trace!(
268               target: "mantle::preconf::dispatch",
269               ?hash, ?reason,
270               "dedup hit; forwarding prior rejection to any pending responder"
271           );
272           fifo.cancel_responder(&hash, reason).await;
```

```
273             return Ok(());
274         }
275     }
```

Listing 2.14: mantle-reth/crates/preconf/src/builder/dispatch.rs

```
420     // (1) Same-sender cascade -Replay entries only. A successor inherits the
421     // predecessor's non-admission outcome; it cannot execute before the
422     // predecessor lands.
423     if source == PreconfSource::Replay &&
424         let Some(kind) = loop_state.sender_blocked_at(&sender, nonce)
425     {
426         match kind {
427             dispatch::BlockKind::Defer => {
428                 metrics::counter!("preconf.fifo.replay_deferred_total").increment(1);
429                 debug!(
430                     target: "mantle::preconf::dispatch",
431                     ?hash, ?sender, nonce,
432                     "replay tx cascade-deferred (predecessor deferred)"
433                 );
434                 return Ok(());
435             }
436             dispatch::BlockKind::Reject => {
437                 // Server pre-apply rejection (predecessor can't land →nonce
438                 // gap) -never handed to the builder, so `Canceled`, not
439                 // `Failed`.
440                 let _ = fifo.mark_canceled(&hash).await;
441                 loop_state.record_excluded(
442                     hash,
443                     PreconfError::BuilderRejected(
444                         "preconf predecessor from same sender rejected (nonce gap)".into(),
445                     ),
446                 );
447                 return Ok(());
448             }
449         }
450     }
451
452     // (2) Block-capacity admission. Reads are `u64` copies →the immutable
453     // borrow of `*info` ends before the `&mut info` closure below.
454     let da_used = info.cumulative_da_bytes_used;
455     let gas_used = info.cumulative_gas_used;
456     match preconf_admission(tx_da, tx_gas, da_used, gas_used, limits, source) {
457         Admission::Admit => {
458             let mut apply_fn =
459                 |tx, h, height| apply_preconf_with_da::<N, _>(builder, info, limits, tx, h, height)
                        ;
460             // Propagate a fatal apply error to abort the whole build; a
461             // per-tx rejection resolves inside `apply_one_preconf` and
462             // returns `Ok(())`.
463             dispatch::apply_one_preconf(fifo, cfg, hash, loop_state, &mut apply_fn).await?;
464         }
465         Admission::Defer => {
```

```
466             loop_state.block_sender(sender, nonce, dispatch::BlockKind::Defer);
467             metrics::counter!("preconf.fifo.replay_deferred_total").increment(1);
468             debug!(
469                 target: "mantle::preconf::dispatch",
470                 ?hash, ?sender, nonce,
471                 "replay tx deferred (transient block capacity); keeping Waiting for next slot"
472             );
473         }
474         Admission::Reject(e) => {
475             loop_state.block_sender(sender, nonce, dispatch::BlockKind::Reject);
476             metrics::counter!("preconf.fifo.da_rejected_total").increment(1);
477             // Server pre-apply capacity rejection (DA / real block gas) -the
478             // tx never reaches the builder, so `Canceled` (like the preconf
479             // block-gas-budget gate), not `Failed` (which means the builder ran
480             // and rejected it).
481             let _ = fifo.mark_canceled(&hash).await;
482             if let Some(resp) = fifo.take_responder(&hash).await {
483                 let _ = resp.send(Err(e.clone()));
484             }
485             loop_state.record_excluded(hash, e);
486         }
487     }
```

Listing 2.15: mantle-reth/crates/preconf/src/builder/payload_builder.rs

Suggestion Move the deduplication check ahead of the admission checks.

2.2.3 Prevent responder overwrites during concurrent preconfirmation retries for the same transaction hash

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When a transaction is submitted through the RPC method `eth_sendRawTransactionWithPreconf()`, the RPC handler invokes function `attach_responder()` to associate a responder with the corresponding `PreconfTxSet` entry. For an entry in the `Timeout`, `Canceled`, or `Failed` state, the function replaces the existing responder without checking whether the responder slot is already occupied. If two requests concurrently retry the same transaction hash, the second request may overwrite and drop the responder registered by the first request. The first request then receives a channel error and invokes the function `cancel_responder()`, which removes the responder currently registered for the second request and sends an error through it. Consequently, both requests may return JSON-RPC errors, even though the pool listener can subsequently invoke the function `push_if_absent()` to restore the entry to the `Waiting` state for execution.

```
677     pub async fn attach_responder(
678         &self,
679         hash: TxHash,
680         origin_instant: Instant,
681         responder: oneshot::Sender<Result<PreconfReceipt, PreconfError>>,
682     ) -> Result<(), AttachError> {
```

```
683         let mut inner = self.inner.lock().await;
684         if let Some(entry) = inner.entries.get_mut(&hash) {
685             match entry.status {
686                 // A prior client already resolved on this hash -the
687                 // apply succeeded and the receipt was delivered via
688                 // `mark_succeeded` + `take_responder`. Any second
689                 // submission would have nothing new to await, so
690                 // surface as `AlreadyInProgress` at the caller.
691                 PreconfStatus::Success => {
692                     return Err(AttachError::AlreadyAttached);
693                 }
694                 // Waiting -the entry is live. Allow attach only when
695                 // no responder is currently registered (fresh listener-
696                 // only push, or the RPC handler that owns the slot has
697                 // taken its responder). If a responder is present, a
698                 // client is actively waiting and we must not overwrite
699                 // its `oneshot::Sender`.
700                 PreconfStatus::Waiting => {
701                     if entry.responder.is_some() {
702                         return Err(AttachError::AlreadyAttached);
703                     }
704                     entry.responder = Some(responder);
705                     return Ok(());
706                 }
707                 // Reclaimable -this is a same-hash retry after a
708                 // `Timeout` (client deadline), `Canceled` (block-gas-budget pre-apply
709                 // reject), or `Failed` (reth builder pre-execute reject;
710                 // tx NOT on chain). Install the fresh responder and
711                 // refresh `inserted_at` so `builder::dispatch`'s
712                 // deadline gate measures against the second submission
713                 // rather than the (already-expired) first. The
714                 // subsequent `push_if_absent` from the pool listener
715                 // flips the entry back to `Waiting` and broadcasts.
716                 PreconfStatus::Timeout | PreconfStatus::Canceled | PreconfStatus::Failed => {
717                     entry.responder = Some(responder);
718                     entry.inserted_at = origin_instant;
719                     return Ok(());
720                 }
721             }
722         }
723         if inner.pending_responders.contains_key(&hash) {
724             return Err(AttachError::AlreadyAttached);
725         }
726         inner.pending_responders.insert(hash, (origin_instant, responder));
727         Ok(())
728     }
```

Listing 2.16: mantle-reth/crates/preconf/src/preconf_tx_set.rs

```
268             // Builder dropped the responder without sending -should not
269             // happen on healthy paths. Mark the entry `Canceled`
270             // (revivable + swept by `clean_reclaimable`) to signal a
271             // server-side failure with the tx never applied.
```

```
272          Some(Err(_recv_err)) => {
273              warn!(target: "mantle::preconf::rpc", ?hash, "responder dropped before send");
274              let _ = self.fifo.mark_canceled(&hash).await;
275              let err = PreconfError::Internal("responder dropped before send".to_string());
276              self.fifo.cancel_responder(&hash, err.clone()).await;
277              Err(preconf_error_to_rpc(&err))
278          }
```

Listing 2.17: mantle-reth/crates/preconf/src/rpc.rs

```
742  pub async fn cancel_responder(&self, hash: &TxHash, err: PreconfError) {
743      let responder = {
744          let mut inner = self.inner.lock().await;
745          let r = inner
746              .entries
747              .get_mut(hash)
748              .and_then(|e| e.responder.take())
749              .or_else(|| inner.pending_responders.remove(hash).map(|(_, r)| r));
750          // Drop any ghost pending responder (invariant #2 violation);
751          // no-op in the normal case.
752          inner.pending_responders.remove(hash);
753          r
754      };
755      if let Some(r) = responder {
756          let _ = r.send(Err(err));
757      }
758  }
```

Listing 2.18: mantle-reth/crates/preconf/src/preconf_tx_set.rs

Suggestion Return the error `AlreadyAttached` whenever any responder slot for the transaction hash is occupied, regardless of the current entry state.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description The `mantle-reth`'s preconfirmation mechanism is a centralized service provided solely by the sequencer, not a protocol-enforced guarantee. The sequencer unilaterally decides which transactions receive preconfirmation eligibility, executes them ahead of ordinary mempool traffic, and returns inclusion promises backed by no bond, slashing, or verifiable proof. A malicious or compromised sequencer can therefore selectively discard commitments, deviate from the acknowledged transaction order, or act on privileged knowledge of preconfirmed transaction contents. Clients relying on preconfirmations bear counterparty risk that the consensus layer neither detects nor penalizes. This design is acceptable only under the assumption that the sequencer remains honest and available.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS