PARADIGM

Reth – Ethereum Execution Client

Security Assessment Report

Version: 2.0

June, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Paradigm Reth codebase. The review focused solely on the security aspects of the Rust implementation of the execution client, though general recommendations and informational comments are also provided.

The review focused solely on the security aspects of the codebase, though general recommendations and informational comments are also provided.

Disclaimer

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

Document Structure

The first section provides an overview of the functionality of the Reth execution client software contained within the scope of the security review.

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

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Reth codebase.

Overview

Reth is an Execution Layer (EL) client implementation for the Ethereum protocol, written in Rust. It is an alternative to other execution clients like Besu, Erigon, Geth and Nethermind, which are responsible for processing and broadcasting transactions, and managing the Ethereum state. Reth is developed with a focus on modularity, user-friendliness and performance.

Users run Reth in combination with a Consensus Layer (CL) client in order to act as a full staking node, and interact with the Ethereum network in a decentralised manner.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Paradigm Reth codebase and first-party dependencies, detailed as follows:

- paradigmxyz/reth 66c9403 (release: v0.2.0-beta.1)
- alloy-rs/trie 4c4f32f (release: v0.3.0)
- alloy-rs/nybbles d294873 (release: v0.2.1)
- alloy-rs/rlp 91fd3d3 (shortly after release v0.3.4)

Retesting activities were performed on the following commits:

- paradigmxyz/reth 560080e (release: v1.0.0-rc.1)
- alloy-rs/trie 80c18f9 (release: v0.4.1)
- alloy-rs/nybbles d294873 (release: v0.2.1)
- alloy-rs/rlp b6a26dc (shortly after release v0.3.5)

The scope of this time-boxed review was strictly limited to files at the commits specified.

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

Approach

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language and Ethereum protocol.

To support this review, the testing team also utilised the following testing tools:

- Clippy linting: `https://doc.rust-lang.org/stable/clippy/index.html`

Output for these automated tools is available upon request.

Additionally, the testing team leveraged fuzz testing techniques (i.e. fuzzing), which is a process allowing the identification of bugs by providing randomised and unexpected data inputs to software with the purpose of causing crashes (in Rust, *panics*) and other unexpected behaviours (e.g. broken invariants,).

Sigma Prime produced fuzzing targets targeting the components using cargo-fuzz, a fuzz-testing framework for Rust software. This framework focuses on:

- **In-process fuzzing**: the fuzzing engine executes the target many times with multiple data inputs in the same process. It must tolerate any kind of input (empty, huge, malformed, etc.);

- **White-box fuzzing**: `cargo-fuzz` leverages compiler instrumentation and requires access to the source code;
- **Coverage-guided fuzzing**: for every input/test case, `cargo-fuzz` tracks code paths (sections of the code reached), and produces variants of each test case to generate additional input data to increase code coverage.

The testing team created a set of fuzzing targets to complement the manual review (see `fuzzing` folder).

Coverage Limitations

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

Findings Summary

The testing team identified a total of 51 issues during this assessment. Categorised by their severity:

- Critical: 4 issues.
- High: 6 issues.
- Medium: 8 issues.
- Low: 20 issues.
- Informational: 13 issues.

Detailed Findings

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

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

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

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

Summary of Findings

ID	Description	Severity	Status
RETH-01	Invalid Side Chain Hashes For State Provider	Critical	Resolved
RETH-02	`TrieUpdates::flush()` Will Change Order Of Operations For Extended Updates	Critical	Resolved
RETH-03	RLP Decoding Allows `CREATE` Transactions For EIP-4844 Types	Critical	Resolved
RETH-04	RLP Decoding Allows Trailing Bytes When Decoding Transactions	Critical	Resolved
RETH-05	State Roots May Not Be Checked For Buffered Blocks	High	Open
RETH-06	`respond_closest()` Shares All Neighbours	High	Resolved
RETH-07	Lack Of Timeout In `EthStream` Handshake	High	Resolved
RETH-08	Reachable `unreachable!()` During Pool Transaction Decoding	High	Resolved
RETH-09	Error messages For EngineAPI May Return An Invalid Hash	High	Resolved
RETH-10	`latest_valid_hash_for_invalid_payload()` Does Not Find Canonical Hashes	High	Resolved
RETH-11	Denial-of-Service Condition Through `PING` Spamming	Medium	Resolved
RETH-12	`find_node` May Be Called With An Invalid Endpoint	Medium	Resolved
RETH-13	Multiplexer Ignores Errors From `P2PStream`	Medium	Resolved
RETH-14	Sub Protocol Messages Are Dropped During `EthStream` Handshake	Medium	Resolved
RETH-15	Arithmetic Overflows In `alloy-nybbles`	Medium	Resolved
RETH-16	Panic When Debug Assertion Is Violated	Medium	Resolved
RETH-17	Bytes May Not Be Nibbles In `From` And `Extend` Traits	Medium	Resolved
RETH-18	`TrieUpdates` Are Not Removed For All Chains During Fork Choice Updates	Medium	Resolved
RETH-19	Unreliable `last_finalized_block_number` initialisation	Low	Resolved
RETH-20	`expect()` Used Without Guarantees Of Success	Low	Resolved
RETH-21	Imported Transactions May Be Removed From Fetcher Without Being Added To The Pool	Low	Resolved
RETH-22	Incorrect Expiration Used For Checking Expired Requests	Low	Resolved
RETH-23	Missing Fields Should Be Skipped When Decoding Messages	Low	Open
RETH-24	Unbounded Channels	Low	Resolved
RETH-25	Connection Denial-of-Service Condition Via Invalid TCP Packets	Low	Open

RETH-26	Out-of-Bounds Access When Reading From Nippy Jar Archives	Low	Resolved
RETH-27	Denial-of-Service Condition Through UDP Spamming	Low	Resolved
RETH-28	Potential Index Out Of Bounds In `kdf()`	Low	Resolved
RETH-29	Arithmetic Overflow In `decrypt_message()`	Low	Resolved
RETH-30	Arithmetic Overflow In `read_body()`	Low	Resolved
RETH-31	Arithmetic Overflow In `new_chain_fork()`	Low	Resolved
RETH-32	`read_header()` Index Out Of Bounds If Input Is Not At Least 32 Bytes	Low	Resolved
RETH-33	RLP Header Is Not Validated In `RequestPair` Decoding	Low	Resolved
RETH-34	RLP Header Is Not Validated In `DisconnectReason`	Low	Resolved
RETH-35	Index Out Of Bounds Panic When Sending Empty Bytes	Low	Resolved
RETH-36	Index Out Of Bounds Panic When Multiplex Message Is Empty	Low	Resolved
RETH-37	Arithmetic Overflows In `ProtocolStream` & `ProtocolProxy`	Low	Resolved
RETH-38	Forks With `next` As Timestamp May Be Confused With Blocknumber	Low	Resolved
RETH-39	`PING`/`PONG` Man-in-the-Middle	Informational	Closed
RETH-40	Database Shrinking Accidentally Enabled	Informational	Resolved
RETH-41	Database Not Opened in Exclusive Mode	Informational	Closed
RETH-42	ENR Responses Are Not Validated	Informational	Resolved
RETH-43	Eclipse Mitigations	Informational	Open
RETH-44	ECIES Protocol Bugs	Informational	Open
RETH-45	Large `base_fee` Overflows Block Base Fee Calculations	Informational	Closed
RETH-46	Missing Documentation for Untrusted `NippyJar` and `Compact` Formatted Data	Informational	Resolved
RETH-47	Missing Panic Comments in `from_compact()`	Informational	Resolved
RETH-48	`is_database_empty()` False Positive For Paths That Are Not Directories	Informational	Resolved
RETH-49	`BlockchainTreeConfig` Concerns Regarding Fixed Finalisation Depth	Informational	Open
RETH-50	Miscellaneous General Comments	Informational	Resolved
RETH-51	Missing Payload Header Validation For Blob Fields, Withdrawals	Informational	Resolved

RETH-01	Invalid Side Chain Hashes For State Provider		
Asset	`reth/crates/blockchain-tree/src/blockchain_tree.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `all_chain_hashes()` may overwrite existing block hashes if the parent chain has overlapping block numbers with the child chains.

```
fn all_chain_hashes(&self, chain_id: BlockChainId) -> BTreeMap<BlockNumber, BlockHash> {
    // find chain and iterate over it,
    let mut chain_id = chain_id;
    let mut hashes = BTreeMap::new();
    loop {
        let Some(chain) = self.state.chains.get(&chain_id) else { return hashes };
        hashes.extend(chain.blocks().values().map(|b| (b.number, b.hash()))); // @audit if there already exists entries for
              ↪  b.number this will overwrite


        let fork_block = chain.fork_block();
        if let Some(next_chain_id) = self.block_indices().get_blocks_chain_id(&fork_block.hash)
        {
            chain_id = next_chain_id;
        } else {
            // if there is no fork block that point to other chains, break the loop.
            // it means that this fork joins to canonical block.
            break
        }
    }
    hashes
}
```

The case which will cause issues is during the second, or higher, iteration of the loop. If the current chain contains a block with `b.number` that is already in the `hashes` mapping, then `hashes` will overwrite the value at key `b.number`.

The impact of this issue is rated as high as these hashes are used inside the EVM for the `BLOCKHASH` opcode. An invalid hash could lead to incorrect values inside the EVM executor and result in an incorrect state root calculation.

Recommendations

To resolve the issue, only insert values in to `hashes` if the block in the parent chain is less than the fork number.

Resolution

The recommendation has been implemented in PR #7669.

RETH-02	`TrieUpdates::flush()` Will Change Order Of Operations For Extended Updates		
Asset	`reth/crates/trie/src/updates.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`TrieUpdates::flush()` will break the order of operations if there are delete and update operations from different blocks.

The following `sort_unstable_by()` command is correct when there is a single block. That is because any delete operations in `StorageTrie` should occur before `StorageNode` updates. However, when there are multiple blocks worth of `trie_operations` then the order may be reversed.

```
let mut trie_operations = Vec::from_iter(self.trie_operations);
trie_operations.sort_unstable_by(|a, b| a.0.cmp(&b.0)); //@audit this sort can break order of operations
for (key, operation) in trie_operations {
    match key {
        TrieKey::AccountNode(nibbles) => match operation {
            TrieOp::Delete => {
                if account_trie_cursor.seek_exact(nibbles)?.is_some() {
                    account_trie_cursor.delete_current()?;
                }
            }
            TrieOp::Update(node) => {
                if !nibbles.0.is_empty() {
                    account_trie_cursor.upsert(nibbles, StoredBranchNode(node))?;
                }
            }
        },
        TrieKey::StorageTrie(hashed_address) => match operation { // @audit always occurs before any `TrieKey::StorageNode` due to
            ↪  sorting
            TrieOp::Delete => {
                if storage_trie_cursor.seek_exact(hashed_address)?.is_some() {
                    storage_trie_cursor.delete_current_duplicates()?;
                }
            }
            TrieOp::Update(..) => unreachable!("Cannot update full storage trie."),
        },
        TrieKey::StorageNode(hashed_address, nibbles) => { // @audit occurs after all `TrieKey::StorageTrie` operations
            if !nibbles.is_empty() {
                // Delete the old entry if it exists.
                if storage_trie_cursor
                    .seek_by_key_subkey(hashed_address, nibbles.clone())?
                    .filter(|e| e.nibbles == nibbles)
                    .is_some()
                {
                    storage_trie_cursor.delete_current()?;
                }

                // The operation is an update, insert new entry.
                if let TrieOp::Update(node) = operation {
                    storage_trie_cursor
                        .upsert(hashed_address, StorageTrieEntry { nibbles, node })?;
                }
            }
        }
    };
}
```

Consider an example where there are two updates `A->B` and `B->C`, each with one operation:

- `A->B` does `Triekey::StorageNode(0xDEAD, 0x11)` with operation `TrieOp::Update(node1)`.
- `B->C` does `TrieKey::StorageTrie(0xDEAD)` with operation `TrieOp::Delete`.

When these operations are processed one block at a time, the `TrieOp::Update` will occur first followed by the `TrieOp::Delete`.

When the updates from `A->B` and `B->C` are combined, the `TrieOp::Delete` will now occur before `TrieOp::Update(node1)`. That is because sort will put all `Triekey::StorageTrie` operations before `Triekey::StorageNode` operations.

Recommendations

Consider modifying `TrieUpdates` such that when `extend()` is called, updates for each block are serialised by block.

That is in the above example updates `A->B` occur before `B->C`.

This can be done by adding a function `TrieUpdates::extend_block()` which stores a `HashMap` for each block rather than combining blocks into a single map.

If trie updates are modified to operate strictly on the previous block, then all block updates may be safely cached in blockchain tree. This solution would require a memory overlay for `AccountsTrie` and `StoragesTrie` which ensures `TrieUpdates` are prepared with respect to previous state rather than some old canonical head.

This solution would be similar to how `BundleStateWithReciepts` works for receipts and reverts, which are stored on a per-block basis and can easily be reverted to the correct block by slicing a vector.

Now `TrieUpdates` would not need to be invalidated on fork choice updates since each update is based strictly on the block before.

Resolution

The issue was quickly resolved by disabling caching for more than one block in PR #7753.

RETH-03	RLP Decoding Allows `CREATE` Transactions For EIP-4844 Types		
Asset	`reth/crates/primitives/src/transaction/eip4844.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

EIP-4844 specifies a restriction that blob transactions must not allow `CREATE` transactions (i.e. those used to deploy bytecode). However, this is achievable in Reth due to the `TransactionKind` field.

The following excerpt is from EIP-4844.

> *The field 'to' deviates slightly from the semantics with the exception that it MUST NOT be 'nil' and therefore must always represent a 20-byte address. This means that blob transactions cannot have the form of a create transaction.*

The following is an excerpt from the `TxEip4844` struct in Reth.

```
pub struct TxEip4844 {
  // ... snipped
  pub to: TransactionKind,
  // ... snipped
}
```

Setting the `to` field as `TransactionKind` allows decoding the address into `TransactionKind::Create()`.

The severity is rated a critical as this is a consensus bug since these transactions are RLP decoded through the EngineAPI `eth_newPayloadV3()` call.

The following is an example test case and data which would allow decoding as a `CREATE` transactsion

```
let data = vec![3, 208, 128, 128, 123, 128, 120, 128, 129, 129, 128, 192, 129, 129, 192, 128, 128, 9];
let res = TransactionSigned::decode_enveloped(&mut &data[..]); // res is Ok() when it should be Err()
```

Recommendations

Change the type of `to` in `TxEip4844` to be `Address` rather than `TransactionKind`.

Alternatively, add a custom implementation of `Decodable` and enforce `to = TransactionKind::Create`.

Resolution

The recommendation has been implemented in PR #8291. The changes ensure that EIP-4844 transactions are not `CREATE` transactions by enforcing the `to` address field be non-empty.

RETH-04	RLP Decoding Allows Trailing Bytes When Decoding Transactions		
Asset	`reth/crates/primitives/src/transaction/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The RLP decoding of transactions in EngineAPI is not strict, it allows excess bytes to be appended to a transaction and still decode successfully.

It is required each set of transaction bytes inside an execution payload to be strictly the number of bytes required to decode the transaction. The following is from the yellow paper.

> 1. *Transaction Execution The execution of a transaction is the most complex part of the Ethereum protocol: it defines the state transition function Y. It is assumed that any transactions executed first pass the initial tests of intrinsic validity. These include: (1) The transaction is well-formed RLP, **with no additional trailing bytes***

If excess bytes are supplied other clients will error. For example Geth will raise an error if there are bytes remaining in the buffer after calling DecodeBytes(). Note here that Geth supports two functions for RLP decoding `Decode()` which allows excess bytes and `DecodeBytes()` which requires strict decoding.

Neither `try_payload_v1_to_block()` or any sub routines will verify the length of each transaction is strict.

try_payload_v1_to_block()

```
27  let transactions = payload
        .transactions
29      .into_iter()
        .map(|tx| TransactionSigned::decode_enveloped(&mut tx.as_ref())) // @audit transaction decoding does not check all bytes are
            ↪ read
31      .collect::<Result<Vec<_>, _>>()?;
    let transactions_root = proofs::calculate_transaction_root(&transactions);
```

`decode_enveloped()` also does not check all data is read, similarly for all sub calls.

decode_enveloped

```
1412 pub fn decode_enveloped(data: &mut &[u8]) -> alloy_rlp::Result<Self> {
        if data.is_empty() {
1414        return Err(RlpError::InputTooShort)
        }
1416
        // Check if the tx is a list
1418    if data[0] >= EMPTY_LIST_CODE {
            // decode as legacy transaction
1420        TransactionSigned::decode_rlp_legacy_transaction(data)
        } else {
1422        TransactionSigned::decode_enveloped_typed_transaction(data)
        }
1424 }
```

The result is a consensus bug as malformed blocks may be accepted on Reth but rejected on other clients. The severity is critical severity due to the ease for an attacker to craft a block with a single malicious transaction.

A minimal case is the bytes input `[201, 3, 56, 56, 128, 43, 36, 27, 128, 3, 192]`. This successfully decodes on Reth `decode_enveloped()` but is rejected on Geth `Transaction::UnmarshalBinary()`.

The issue of allowing excess bytes in RLP may occur throughout the code base. An alternate example is referenced in load_and_reinsert_transactions(). Other cases do not raise consensus bugs as they do not occur on calls from the consensus layer client, however they could cause issues in networking with some transactions being accepted by Reth and rejected by other clients.

Recommendations

Modify `decode_enveloped()` or `try_payload_v1_to_block()` to reject transactions with excess bytes.

Furthermore, add the function `decode_exact()` to the `Decodable` trait such that there is an alternative decoding method which will perform length checks on the buffer.

Resolution

Additional checks have been added in PR #8296. These checks require a strict equality between the RLP payload length and the number of bytes provided.

RETH-05	State Roots May Not Be Checked For Buffered Blocks		
Asset	`reth/crates/blockchain-tree/src/blockchain_tree.rs`		
Status	**Open**		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When a new payload is added to the chain, it is checked to see if any children exist in the buffer blocks as these may be connected to the chain. Upon connecting the buffered blocks, state root validation does not occur for the buffered blocks. Furthermore, state root validation does not occur later when making the blocks canonical unless the block is also the new canonical tip. As a result, blocks may be added to the chain with invalid state roots.

The function `try_connect_buffered_blocks()` will allow buffered blocks to skip state root validation. This function is called when adding a new payload and may result in blocks being inserted into side chains.

blockchain_tree.rs

```
828 fn try_connect_buffered_blocks(&mut self, new_block: BlockNumHash) {
        trace!(target: "blockchain_tree", ?new_block, "try_connect_buffered_blocks");
830
        // first remove all the children of the new block from the buffer
832     let include_blocks = self.state.buffered_blocks.remove_block_with_children(&new_block.hash);
        // then try to reinsert them into the tree
834     for block in include_blocks.into_iter() {
            // dont fail on error, just ignore the block.
836         let _ = self
                .try_insert_validated_block(block, BlockValidationKind::SkipStateRootValidation) // @audit state root validation is
                    ↪ skipped
838             .map_err(|err| {
                    debug!(
840                     target: "blockchain_tree", %err,
                        "Failed to insert buffered block",
842                 );
                    err
844             });
        }
846 }
```

Note that pending blocks are assumed to be “fully validated” in other parts of the code, however, this is not the case if they were originally buffered as the state root is not validated.

Later, if a forkchoice update sets a number of blocks as canonical, it will only perform state root checks on the tip. This can be seen in commit_canonical_to_database() which performs the checks.

The issue is that blocks strictly in between the new tip and the old tip do not have their state roots validated. Therefore, each block header except the tip may contain any arbitrary value for `state_root`, if it was previously buffered.

The impact is high as it may result in blocks with arbitrary state roots being accepted, which is a consensus bug and may allow malicious state proofs for light clients and external applications.

Recommendations

A solution is to use exhaustive validation when attempting to insert buffered blocks in `try_insert_validated_block()`. This will increase the processing time as state roots must be checked; however, it will prevent potentially invalid blocks being added to the chain.

Resolution

The development team have opted to delay fixing this issue.

The recommendation suggests using exhaustive validation, however this has a significant degradation in performance. PR #8026 implements these changes but was not merged due to the performance impact.

A follow-up PR #8128, which is not yet merged, contains changes to calculate the storage root in parallel rather than single threaded. The issue is marked as open while development continues to resolve the issue.

RETH-06	`respond_closest()` Shares All Neighbours		
Asset	`reth/crates/net/discv4/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The function respond_closest() is intended to return the 16 closest nodes from a target. However, the current implementation will return the entire local DHT.

The following line does not include a `take()` and therefore will return all of `closest_values()` which represents the entire DHT sorted by distance.

respond_closest()

```
1374 fn respond_closest(&mut self, target: PeerId, to: SocketAddr) {
         let key = kad_key(target);
1376     let expire = self.send_neighbours_expiration();
         let all_nodes = self.kbuckets.closest_values(&key).collect::<Vec<_>>(); // @audit does not take() the correct number of nodes
1378
         for nodes in all_nodes.chunks(SAFE_MAX_DATAGRAM_NEIGHBOUR_RECORDS) {
1380         let nodes = nodes.iter().map(|node| node.value.record).collect::<Vec<NodeRecord>>();
             trace!(target: "discv4",  len = nodes.len(), to=?to,"Sent neighbours packet");
1382         let msg = Message::Neighbours(Neighbours { nodes, expire });
             self.send_packet(msg, to);
1384     }
     }
```

The impact is significant as other Reth nodes will reject the neighbours packets since they will generally exceed the `MAX_NODES_PER_BUCKET` count. As seen in the following check in `on_neightbours()`.

on_neighbours()

```
1293 let total = request.response_count + msg.nodes.len();
1295
     // Neighbours response is exactly 1 bucket (16 entries).
1297 if total <= MAX_NODES_PER_BUCKET { // @audit will reject messages with too many entries
         request.response_count = total;
1299 } else {
         trace!(target: "discv4", total, from=?remote_addr,  "Received neighbors packet entries exceeds max nodes per bucket");
1301     return
     }
```

Recommendations

Add `take()` to the iterator before calling `collect()`. This will ensure only the correct number of nodes is used.

Resolution

The issue has been resolved in PR #6842 by adding `take(MAX_NODES_PER_BUCKET)`.

RETH-07	Lack Of Timeout In `EthStream` Handshake		
Asset	`reth/crates/net/eth-wire/src/ethstream.rs, crates/net/ecies/src/stream.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

There is no timeout when waiting for a reply status message in `UnauthedEthStream`. An attacker could abuse this by opening countless connections and performing the handshake up to the status message of the `EthStream`. As a result, each of the spawned threads would be indefinitely pending, waiting for a status message.

handshake()

```
54 pub async fn handshake(
       mut self,
56     status: Status,
       fork_filter: ForkFilter,
58 ) -> Result<(EthStream<S>, Status), EthStreamError> {
       trace!(
60         %status,
           "sending eth status to peer"
62     );

64     // we need to encode and decode here on our own because we don't have an `EthStream` yet
       // The max length for a status with TTD is: <msg id = 1 byte> + <rlp(status) = 88 byte>
66     let mut our_status_bytes = BytesMut::with_capacity(1 + 88);
       ProtocolMessage::from(EthMessage::Status(status)).encode(&mut our_status_bytes);
68     let our_status_bytes = our_status_bytes.freeze();
       self.inner.send(our_status_bytes).await?;
70
       let their_msg_res = self.inner.next().await; //@audit lack of timeout here
72
       let their_msg = match their_msg_res {
74         Some(msg) => msg,
           None => {
76             self.inner.disconnect(DisconnectReason::DisconnectRequested).await?;
               return Err(EthStreamError::EthHandshakeError(EthHandshakeError::NoResponse))
78         }
       }?;
```

The unbounded read occurs in `self.inner.next().await`. The call will watch the TCP stream and wait for a message to be received. If a malicious user on the other end of the connection does not send a message the thread will block indefinitely.

As a result the total number of spawned threads is increased and these threads are permanently blocking, which limits the number of possible connections.

The same bug is also present in the ECIES handshake for both incoming and outgoing connections as it uses `transport.try_next().await`.

connect()

```rust
pub async fn connect(
    transport: Io,
    secret_key: SecretKey,
    remote_id: PeerId,
) -> Result<Self, ECIESError> {
    let ecies = ECIESCodec::new_client(secret_key, remote_id)
        .map_err(|_| io::Error::new(io::ErrorKind::Other, "invalid handshake"))?;

    let mut transport = ecies.framed(transport);

    trace!("sending ecies auth ...");
    transport.send(EgressECIESValue::Auth).await?;

    trace!("waiting for ecies ack ...");

    let msg = transport.try_next().await?; // @audit no timeout
```

incoming()

```rust
pub async fn incoming(transport: Io, secret_key: SecretKey) -> Result<Self, ECIESError> {
    let ecies = ECIESCodec::new_server(secret_key)?;

    trace!("incoming ecies stream");
    let mut transport = ecies.framed(transport);
    let msg = transport.try_next().await?; // @audit no timeout
```

Recommendations

The issue may be mitigated by wrapping each of the `next()` and `try_next()` calls mentioned in the description with a `tokio` timeout.

```rust
tokio::time::timeout(HANDSHAKE_TIMEOUT, self.inner.next()).await
```

Resolution

The issue has been resolved by adding a high level timeout which covers the entire handshake. Changes can be seen in PR #7219.

It is recommended to also include timeouts over each individual transport layer read. This will increase the robustness, protecting from future modifications to the code. Additionally, it will give more granularity in debugging.

RETH-08	Reachable `unreachable!()` During Pool Transaction Decoding		
Asset	`crates/reth/primitives/src/transaction/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There is a reachable `unreachable!()` macro that will cause a panic when decoding `PooledTransactionsElement()`.

The function `decode_enveloped()` will take raw bytes as input and decode them into a transaction depending on the variant. If a transaction has the first byte as larger than 192, it can be considered a legacy transaction since it is an RLP encoded list. Otherwise it will be decoded as an EIP-4844, EIP-2930 or EIP-1559 transaction type.

decode_enveloped()

```
1314    pub fn decode_enveloped(data: &mut &[u8]) -> alloy_rlp::Result<Self> {
        if data.is_empty() {
1316        return Err(RlpError::InputTooShort)
        }
1318
        // Check if the tx is a list - tx types are less than EMPTY_LIST_CODE (0xc0)
1320    if data[0] >= EMPTY_LIST_CODE {
            // decode as legacy transaction
1322        let (transaction, hash, signature) =
                TransactionSigned::decode_rlp_legacy_transaction_tuple(data)?;
1324
            Ok(Self::Legacy { transaction, signature, hash })
1326    } else {
            // decode the type byte, only decode BlobTransaction if it is a 4844 transaction
1328        let tx_type = *data.first().ok_or(RlpError::InputTooShort)?;

1330        if tx_type == EIP4844_TX_TYPE_ID {
                // ... snipped
1332            let blob_tx = BlobTransaction::decode_inner(data)?;
                Ok(PooledTransactionsElement::BlobTransaction(blob_tx))
1334        } else {
                // DO NOT advance the buffer for the type, since we want the enveloped decoding to
1336            // decode it again and advance the buffer on its own.
                let typed_tx = TransactionSigned::decode_enveloped_typed_transaction(data)?; // @audit can be called with `data[0]
                    ↪  = 0
1338
                // ... snipped
1340        }
        }
1342    }
```

There is a subroutine to `decode_enveloped_typed_transaction()` which is intended to decode each of EIP-2930, EIP-1559 and EIP-4844 transaction types but should not be called for legacy transactions.

decode_enveloped_typed_transaction()

```
1244 pub fn decode_enveloped_typed_transaction(
         data: &mut &[u8],
1246     ) -> alloy_rlp::Result<TransactionSigned> {
         // keep this around so we can use it to calculate the hash
1248     let original_encoding_without_header = *data;

1250

         let tx_type = *data.first().ok_or(RlpError::InputTooShort)?;
1252     data.advance(1);

1254

         // decode the list header for the rest of the transaction
1256     let header = Header::decode(data)?;
         if !header.list {
1258         return Err(RlpError::Custom("typed tx fields must be encoded as a list"))
         }
1260

1262     let remaining_len = data.len();

1264

         // length of
```

Now `data[0]` represents the transaction type, with the value zero used to represent legacy. Hence, if we set `data[0] = 0` it will cause `tx_type = TxType::Legacy` and thus cause a panic.

Calling `PooledTransactionsElement::decode_enveloped()` with the following input will trigger the *"unreachable"* panic.

```
let data = vec![0, 195, 139, 10, 171, 171, 171, 171];
PooledTransactionsElement::decode_enveloped(&data); // @audit panics
```

Recommendations

Return an error if `tx_type = TxType::Legacy` in `decode_enveloped_typed_transaction()` rather than `unreachable!()`.

Resolution

The recommendation has been implemented in PR #7339. An error will be returned in place of the `unreachable!()`.

RETH-09	Error messages For EngineAPI May Return An Invalid Hash		
Asset	`reth/crates/consensus/beacon/src/engine/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The function `prepare_invalid_reponse()` does not ensure the status message contains a valid `latest_valid_hash`.

The case arises when two invalid blocks occur in a row. The invalid `header.parent_hash` is passed as a parameter to `prepare_invalid_response()`, this parent hash is presently unvalidated, and as such may be invalid.

check_invalid_ancestor_with_head()

```
756 fn check_invalid_ancestor_with_head(
        &mut self,
758     check: B256,
        head: B256,
760 ) -> Option<PayloadStatus> {
        // check if the check hash was previously marked as invalid
762     let header = self.invalid_headers.get(&check)?;

764     // populate the latest valid hash field
        let status = self.prepare_invalid_response(header.parent_hash); // @audit this parent_hash is not ensured to be valid
766
        // insert the head block into the invalid header cache
768     self.invalid_headers.insert_with_invalid_ancestor(head, header);

770     Some(status)
    }
```

prepare_invalid_response()

```
736 fn prepare_invalid_response(&self, mut parent_hash: B256) -> PayloadStatus {
        // Edge case: the `latestValid` field is the zero hash if the parent block is the terminal
738     // PoW block, which we need to identify by looking at the parent's block difficulty
        if let Ok(Some(parent)) = self.blockchain.header_by_hash_or_number(parent_hash.into()) {
740         if !parent.is_zero_difficulty() {
                parent_hash = B256::ZERO;
742         }
        }
744
        PayloadStatus::from_status(PayloadStatusEnum::Invalid {
746         validation_error: PayloadValidationError::LinksToRejectedPayload.to_string(),
        })
748     .with_latest_valid_hash(parent_hash) // @audit the unvalidated parent hash is attached here
    }
```

One such case where this may occur in the CL layer is when a block A is missing and the child block B is sent to the EL. Assuming both blocks A and B are invalid. If two calls to `engine_newPayload(block_B)` are made then the second call will return status `INVALID` with `latest_valid_hash = block_A`. However, block A has not been received or validated.

The impact is rated as high as it is a consensus bug and CL implementations use the `last_valid_hash` as part of the fork choice.

Recommendations

The helper function `latest_valid_hash_for_invalid_payload()` may be used in `prepare_invalid_response()` to determine the latest valid hash or set it to `None` if it cannot be determined.

Resolution

PR #8123 iterates through invalid ancestors to search for the last valid hash. Only when a valid hash is found will it be returned. Otherwise, `None` will be returned.

RETH-10	`latest_valid_hash_for_invalid_payload()` Does Not Find Canonical Hashes		
Asset	`reth/crates/consensus/beacon/src/engine/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The function `latest_valid_hash_for_invalid_payload()` will return `None` if the `parent_hash` is canonical.

latest_valid_hash_for_invalid_payload()

```
703 fn latest_valid_hash_for_invalid_payload(
        &self,
705     parent_hash: B256,
        insert_err: Option<&InsertBlockErrorKind>,
707 ) -> Option<B256> {
        // check pre merge block error
709     if insert_err.map(|err| err.is_block_pre_merge()).unwrap_or_default() {
            return Some(B256::ZERO)
711     }

713     // If this is sent from new payload then the parent hash could be in a side chain, and is
        // not necessarily canonical
715     if self.blockchain.header_by_hash(parent_hash).is_some() { // @audit returns None if parent_hash is canonical
            // parent is in side-chain: validated but not canonical yet
717         Some(parent_hash)
        } else {
719         let parent_hash = self.blockchain.find_canonical_ancestor(parent_hash)?; // @audit returns None if parent_hash is canonical
            let parent_header = self.blockchain.header(&parent_hash).ok().flatten()?;
721
            // we need to check if the parent block is the last POW block, if so then the payload is
723         // the first POS. The engine API spec mandates a zero hash to be returned:
            if !parent_header.is_zero_difficulty() {
725             return Some(B256::ZERO)
            }
727
            // parent is canonical POS block
729         Some(parent_hash)
        }
731 }
```

Within the `latest_valid_hash_for_invalid_payload()` function, the first subroutine `self.blockchain.header_by_hash()` will only return `Some` if the `parent_hash` is within a side chain. Therefore, if it is canonical, `None` will be returned.

The first case will search for blocks within a side chain and the `else` statement is intended to find the canonical blocks. However, `self.blockchain.find_canonical_ancestor(parent_hash)` will also return `None` if `parent_hash` is not in a side chain.

Hence, if `parent_hash` is a canonical block, the function `latest_valid_hash_for_invalid_payload()` will return `None`.

The specification expects a valid hash to be returned if it can be found and only return `None` when the chain is disconnected.

Recommendations

To resolve this issue, update the `else` statement such that it will search for canonical blocks.

Furthermore, consider updating the code to handle the case where the `parent_header` is in a set of buffered blocks leading to an invalid header, as the `InvalidHeaderCache` may contain the required header.

This requires checking each parent hash to see if it exists in the canonical chain, a side chain, the buffered blocks or invalid headers cache and continuing until a fully validated block is found, or the chain is disconnected.

Resolution

The first portion of the recommendations have been implemented in PR #7716. The updates will search for canonical blocks in the database as well as non-canonical blocks in side chains.

RETH-11	Denial-of-Service Condition Through `PING` Spamming		
Asset	`crates/reth/net/discv4/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There is an unbounded vector which will be appended to for each outgoing PING once the maximum number of pending PINGs has been reached. This queue will be appended to for each incoming PING.

```
fn on_ping(&mut self, ping: Ping, remote_addr: SocketAddr, remote_id: PeerId, hash: B256) {

    // ... snipped

    let mut is_new_insert = false;
    let mut needs_bond = false;
    let mut is_proven = false;

    let old_enr = match self.kbuckets.entry(&key) {

        // ... snipped

        kbucket::Entry::Absent(entry) => {
            let mut node = NodeEntry::new(record);
            node.last_enr_seq = ping.enr_sq;

            match entry.insert(..) {
                BucketInsertResult::Inserted | BucketInsertResult::Pending { .. } => {
                    // mark as new insert if insert was successful
                    is_new_insert = true;
                }
                BucketInsertResult::Full => {
                    // ... snipped
                    needs_bond = true;
                }
                BucketInsertResult::TooManyIncoming | BucketInsertResult::NodeExists => {
                    needs_bond = true;
                    // insert unsuccessful but we still want to send the pong
                }
                BucketInsertResult::FailedFilter => return,
            }

            None
        }
        kbucket::Entry::SelfEntry => return,
    };

    // send the pong first, but the PONG and optionally PING don't need to be send in a
    // particular order
    let pong = Message::Pong(Pong {
        // ... snipped
    });
    self.send_packet(pong, remote_addr);

    // if node was absent also send a ping to establish the endpoint proof from our end
    if is_new_insert {
        self.try_ping(record, PingReason::InitialInsert);
    } else if needs_bond {
        self.try_ping(record, PingReason::EstablishBond); // @audit will be called for BucketInsertResult TooManyIncoming or Full
```

The attack is for a malicious node to spam new PING packets to a node. Each ping will be processed and a PONG

message returned. Furthermore, if there are too many incoming nodes, the insert fails and the `needs_bond` field will be set to true and `try_ping()` will be called.

Since, `try_ping()` contains the unbounded vector `queued_pings` which is appended to once the pending pings are full. As it is cheap for a malicious node with a large network bandwidth to spam PING packets, it would be possible to reach the maximum pending PINGs and then expand the `queued_pings` indefinitely.

The impact of the issue is two-fold:

- it could cause an OOM (out of memory) crash if the vector grows sufficiently large; and
- valid PINGs will not be processed until the queue is emptied.

Recommendations

It is recommended to drop out-going PING requests if there is insufficient bandwidth to process requests.

Resolution

A limit to the number of PING requests has been added in PR #7999. The change prevents unbounded growth of PING requests.

RETH-12	`find_node` May Be Called With An Invalid Endpoint		
Asset	`crates/reth/net/discv4/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

There is a `CAUTION` comment over the `find_node()` function which can be reached.

```
/// Sends a new `FindNode` packet to the node with `target` as the lookup target.
///
/// CAUTION: This expects there's a valid Endpoint proof to the given `node`.
fn find_node(&mut self, node: &NodeRecord, ctx: LookupContext) {
    trace!(target: "discv4", ?node, lookup=?ctx.target(), "Sending FindNode");
    ctx.mark_queried(node.id);
    let id = ctx.target();
    let msg = Message::FindNode(FindNode { id, expire: self.find_node_expiration() });
    self.send_packet(msg, node.udp_addr());
    self.pending_find_nodes.insert(node.id, FindNodeRequest::new(ctx));
}
```

It is possible to call `find_node()` for a node record which does not yet have a validated endpoint.

The case can be reached if the other user has sent a PING but not sent a PONG response. The node record will be added to the DHT, and return an entry in `self.kbuckets.entry()`.

The function `on_neighbours()` will then iterate through all entries including those with `has_endpoint_proof = false` and if this node is the closest it will reach out to that node via `find_node()`.

```
for closest in closest {
    let key = kad_key(closest.id);
    match self.kbuckets.entry(&key) {
        BucketEntry::Absent(entry) => {
            // the node's endpoint is not proven yet, so we need to ping it first, on
            // success, we will add the node to the pending_lookup table, and wait to send
            // back a Pong before initiating a FindNode request.
            // In order to prevent that this node is selected again on subsequent responses,
            // while the ping is still active, we always mark it as queried.
            ctx.mark_queried(closest.id);
            let node = NodeEntry::new(closest);
            match entry.insert(
                node,
                NodeStatus {
                    direction: ConnectionDirection::Outgoing,
                    state: ConnectionState::Disconnected,
                },
            ) {
                BucketInsertResult::Inserted | BucketInsertResult::Pending { .. } => {
                    // only ping if the node was added to the table
                    self.try_ping(closest, PingReason::Lookup(closest, ctx.clone()))
                }
                BucketInsertResult::Full => {
                    // new node but the node's bucket is already full
                    self.notify(DiscoveryUpdate::DiscoveredAtCapacity(closest))
                }
                _ => {}
            }
        }
        BucketEntry::SelfEntry => {
            // we received our own node entry
        }
        _ => self.find_node(&closest, ctx.clone()), // @audit does not check has_endpoint_proof
    }
}
```

Therefore, it has called `find_node()` with an unvalidated endpoint.

The impact is rated as low as it cannot be determined what the effect of calling `find_node()` with an invalid endpoint would be. However, the likelihood is high as this attack can be exploited by any node on the network.

Recommendations

To resolve this issue, in `on_neighbours()` first check that the closest entry has `has_endpoint_proof = true` before calling `find_node()` for that endpoint.

Resolution

PR #8002 implements the suggestion mitigation.

RETH-13	Multiplexer Ignores Errors From `P2PStream`		
Asset	`reth/crates/net/eth-wire/src/multiplex.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When the function `RplxSatelliteStream::poll_next()` is polled, it will call `poll_ready()` for the internal `P2PStream` sink to determine if there is sufficient space in the buffer to send messages. If there is space, then messages will be added to the `P2PStream` sink.

However, for the case where `P2PStream::poll_ready()` returns `Poll::Ready(Err(err))`, the error is ignored.

```
loop {
    match this.inner.conn.poll_ready_unpin(cx) {
        Poll::Ready(_) => { //@audit error case is ignored
            if let Some(msg) = this.inner.out_buffer.pop_front() {
                if let Err(err) = this.inner.conn.start_send_unpin(msg) {
                    return Poll::Ready(Some(Err(err.into())))
                }
            } else {
                break
            }
        }
        Poll::Pending => {
            conn_ready = false;
            break
        }
    }

}
```

`P2PStream::poll_ready()` will have a return value `Poll<Result<(), P2PStream::Error>>`. In this match statement, the error case is treated the same as `Ok(())`: it will pop the front of `out_buffer` and attempt to send it through the stream.

Recommendations

Handle the error case for `P2PStream::poll_ready()`. It could be desirable here to close the connection on error.

Resolution

The development team have modified the stream to begin disconnecting if an error occurs in the inner stream. Changes can be seen in PR #7988.

RETH-14	Sub Protocol Messages Are Dropped During `EthStream` Handshake		
Asset	`reth/crates/net/eth-wire/src/multiplex.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The multiplexer will drop messages from the non-primary protocol during the primary protocol handshake.

```
let (mut multiplex_stream, their_status) = RlpxProtocolMultiplexer::new(p2p_stream)
    .into_eth_satellite_stream(status, fork_filter) //@audit handshake occurs here
    .await
    .unwrap();

// install additional handlers
for handler in extra_handlers.into_iter() {
    let cap = handler.protocol().cap;
    let remote_peer_id = their_hello.id;
    multiplex_stream
        .install_protocol(&cap, move |conn| { //@audit additional protocols added here
            handler.into_connection(direction, remote_peer_id, conn)
        })
        .ok();
}
```

The handshake occurs in `into_eth_satellite_stream()`. During this procedure, any messages not related to the primary protocol are delegated to the associated sub protocol via `delegate_message()`.

However, there is the limitation that `delegate_message()` will not yet contain any items in `self.protocols` since `install_protocol()` has not been called.

```
fn delegate_message(&mut self, cap: &SharedCapability, msg: BytesMut) -> bool {
    for proto in &self.protocols { // @audit empty list during the handshake
        if proto.shared_cap == *cap {
            proto.send_raw(msg);
            return true
        }
    }
    false
}
```

The messages are not cached and so are silently dropped.

Recommendations

It is recommended to either cache the sub protocol messages during the primary handshake or to install the sub protocols before initiating the handshake.

Resolution

The issue is resolved in PR #9086.

RETH-15	Arithmetic Overflows In `alloy-nybbles`		
Asset	`alloy-nybbles/src/nibbles.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`get_bytes()`

The function `get_bytes()` will overflow with input `i = usize::MAX`, leading to memory out of bounds reads.

```
pub fn get_byte(&self, i: usize) -> Option<u8> {
    if i + 1 < self.len() {
        Some(unsafe { self.get_byte_unchecked(i) })
    } else {
        None
    }
}
```

`get_byte_unchecked()` does not validate that it reads within the bounds of the `SmallVec`. Thus, if we pass `i = usize::MAX` it will read some undefined memory.

The impact here is significant as it is conceivable that a malicious node may attempt to call `get_byte()` with a large value.

`unpack_heap()`

There is a potential overflow in the function `unpack_heap()` if a significantly large slice is passed as `data`. This is not feasible on 64 bit architecture however, if `usize` is 32 bits this may be reachable.

```
fn unpack_heap(data: &[u8]) -> Self {
    // Collect into a vec directly to avoid the smallvec overhead since we know this is going on
    // the heap.
    debug_assert!(data.len() > 32);
    let unpacked_len = data.len() * 2; // @audit potential overflow if data.len() is large
    let mut nibbles = Vec::with_capacity(unpacked_len);
    // SAFETY: enough capacity.
    unsafe { Self::unpack_to_unchecked(data, nibbles.as_mut_ptr()) };
    // SAFETY: within capacity and `unpack_to` initialized the memory.
    unsafe { nibbles.set_len(unpacked_len) };
    // SAFETY: the capacity is greater than 64.
    unsafe_assume!(nibbles.capacity() > 64);
    Self(SmallVec::from_vec(nibbles))
}
```

Recommendations

Add checked maths for these instances and return a `None` or error if there is an overflow.

Resolution

The recommendations have been implemented in PRs #4 and #6.

RETH-16	Panic When Debug Assertion Is Violated		
Asset	`crates/net/network/src/transactions/fetcher.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

In debug builds of Reth, when invoking the client via the CLI via `reth node`, the client will panic due to a violation of a debug assertion on line [**668-681**] of `crates/net/network/src/transactions/fetcher.rs`.

```
Error: Critical task `p2p txpool` panicked: ``%hash` in `@buffered_hashes` that's not in
      ↪  `@hashes_fetch_inflight_and_pending_fetch`, `@buffered_hashes` should be a subset of keys in
      ↪  `@hashes_fetch_inflight_and_pending_fetch`, broken invariant `@buffered_hashes` and
      ↪  `@hashes_fetch_inflight_and_pending_fetch`,
`%hash`: 0x5ec3b4dc46dbcc15935dd390de5bcd2050c865349a292187e99db944627d6076`
```

This debug assertion exists to ensure the following:

> *"The set of buffered hashes and the set of hashes currently being fetched are not disjoint.""*

Recommendations

Convert the debug assertion to a debug-level log or otherwise consider the codepath causing this assertion to fail.

Resolution

The recommendation has been implemented in commit 5bd2d34.

RETH-17	Bytes May Not Be Nibbles In `From` And `Extend` Traits		
Asset	`alloy-nybbles/src/nibbles.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The traits for `Extend<u8>` and `From<Vec<u8>>` do not ensure the incoming bytes fit within the nibble range `0x00` to `0x0F`.

Nibbles are 4 bit objects which are stored as bytes in the underlying Rust structs. Two traits have been added which append raw bytes to nibbles. However, there are no range checks on these traits to ensure the incoming bytes each fall in the range `0x00` to `0xF0`.

Crates should generally prevent users from incorrectly using public functions and traits. Instead, there is the function `from_nibbles_unchecked()` which can be used to convert a slice to nibbles.

Recommendations

It is recommended to remove the `Extend<u8>` and `From<Vec<u8>>` trait implementations.

For `From<Vec<u8>>` calls can be replaced with `from_nibbles_unchecked()` if the underlying slice is in nibble format. Note that it may be clearer to rename this function `from_slice_unchecked()` and stating in the doc comments there is no unpacking.

For `Extend<u8>` this can be replaced with the unchecked function `extend_from_slice()`. Note it may be worth renaming this `extend_from_slice_unchecked()` to state it does not perform unpacking. Lastly, the doc comments could be updated to state it takes a slice and does not perform unpacking.

Resolution

PR #8 resolves the issue through a number of changes:

- remove `From<Vec<u8>>` impl in favour of `from_vec_unchecked()` method
- remove `Extend` impl in favour of `extend_from*` methods
- allow unchecked access to underlying `SmallVec`
- add "safe" counterparts to `_unchecked` methods that check validity of nibbles, and vice versa add missing checks and unchecked versions
- update docs to use safe versions, add examples of bad usage

RETH-18	`TrieUpdates` Are Not Removed For All Chains During Fork Choice Updates		
Asset	`reth/crates/blockchain-tree/src/blockchain_tree.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`TrieUpdates` are not removed from all chains when a fork choice update occurs.

Each `TrieUpdates` is based off a database snapshot at a certain block number. When a fork choice update is made, database operations occur to the `AccountsTrie` and `StoragesTrie` tables. The changes invalidate any `TrieUpdates` stored in memory.

The function `make_canonical()` does not invalidate `TrieUpdates` for chains other than the current one.

Consider a scenario where a canonical block `A` is a fork block with two children blocks `B` and `C`. Each of blocks `B` and `C` will cache the `TrieUpdates` on new payload calls. Say a fork choice update occurs which makes block `B` the new canonical head. The `TrieUpdates` for the chain containing block `C` will not be removed.

Recommendations

To mitigate this issue consider iterating through all chains in the blockchain tree and removing any `TrieUpdates` whenever the canonical head changes in `make_canonical()`.

Resolution

PR #8370 removes all cached `TrieUpdates` during `make_canonical()`.

RETH-19	Unreliable `last_finalized_block_number` initialisation		
Asset	`reth/crates/blockchain-tree/src/blockchain_tree.rs` & `reth/crates/blockchain-tree/src/block_indices.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In `BlockchainTree::new()`, the `last_finalized_block_number` value is initialised to a fixed depth value based on `BlockchainTreeConfig::max_reorg_depth`, which does not correspond to the actual finality depth[1] as last reported by the consensus layer (CL). When the Ethereum network is subject to severe, adverse conditions, there are no fixed bounds to finality depth and the actual finalised block can be earlier than the initialised `last_finalized_block_number`. This can break internal assumptions and leave the `BlockchainTree` in an inconsistent, unexpected state.

```
impl<DB: Database, EVM: ExecutorFactory> BlockchainTree<DB, EVM> {
    /// Create a new blockchain tree.
    pub fn new(
        externals: TreeExternals<DB, EVM>,
        config: BlockchainTreeConfig,
        prune_modes: Option<PruneModes>,
    ) -> RethResult<Self> {
        let max_reorg_depth = config.max_reorg_depth() as usize; // @audit set to 64 in Reth
        // The size of the broadcast is twice the maximum reorg depth, because at maximum reorg
        // depth at least N blocks must be sent at once.
        let (canon_state_notification_sender, _receiver) =
            tokio::sync::broadcast::channel(max_reorg_depth * 2);

        let last_canonical_hashes =
            externals.fetch_latest_canonical_hashes(config.num_of_canonical_hashes() as usize)?; // @audit based on
                ↪  max_reorg_depth, set to fixed 256 value

        // TODO(rakita) save last finalized block inside database but for now just take
        // `tip - max_reorg_depth`
        // https://github.com/paradigmxyz/reth/issues/1712
        let last_finalized_block_number = if last_canonical_hashes.len() > max_reorg_depth {
            // we pick `Highest - max_reorg_depth` block as last finalized block.
            last_canonical_hashes.keys().nth_back(max_reorg_depth)
        // ... snipped (error handling)
        Ok(Self {
            externals,
            state: TreeState::new(
                last_finalized_block_number,
                last_canonical_hashes,
                config.max_unconnected_blocks(),
            ),
            // ... snipped
        })
    }
```

The source of this issue was previously noted and raised internally as reth#1712 (which appears to have been automatically closed due to inactivity) but the impact may not have been thoroughly evaluated.

As specific, external exploits have not been identified, this issue has been deemed of *medium* impact. The testing team notes that `max_reorg_depth` is only currently used on initialisation and a running Reth node can handle unbounded finality depth and reorgs greater than `max_reorg_depth` (as long as the non-finality period begins after startup). These

[1] Finality depth is the distance between the current head of the chain and the most recent finalised block.

issues would not be encountered during normal network conditions but are vital to the safety of the protocol in situations where correct operation of individual nodes is most important.

Detail and Analysis

After initialisation, these values are saved in the `BlockIndices` `last_finalized_block` and `canonical_chain` fields. On startup, the first Engine API message sent by the CL is `engine_forkchoiceUpdated` [2]

There are two scenarios of interest, where this issue can cause unexpected behaviour. They both involve Reth being started during a long period where the network has been unable to finalise, such that the first `engine_forkchoiceUpdated` contains a `finalizedBlockHash` corresponding to a block with a number *less* than the initialised `BlockIndices::last_finalized_block`. After validation, the main processing for this message is done by `BeaconConsensusEngine::on_fork_choice_updated()`.

Scenario A: Quick restart

Reth was quite recently running, such that the `engine_forkchoiceUpdated` message's `headBlockHash` was already in Reth's canonical chain.

- From the perspective of the `BlockchainTree`, `make_canonical` is first executed. This returns `Ok(CanonicalOutcome::AlreadyCanonical { header })`
- The RPC message contains no `PayloadAttributes`, so the `BeaconConsensusEngine` next executes `self.ensure_consistent_state_with_status()` at line [443]. This passes through to `self.update_finalized_block()` and `BlockchainTree::finalize_block()`
- `BlockchainTree::finalize_block()` then calls through to `BlockIndices::finalize_canonical_blocks`.
- `BlockIndices::finalize_canonical_blocks` (shown below) is executed with a `finalized_block` argument less than `self.last_finalized_block`.

[2] This offers a reasonable explanation https://hackmd.io/@danielrachi/engine_api#Node-startup. Though it appears that this order is observed rather than directly specified.

```
328 /// Used for finalization of block.
    ///
330 /// Return list of chains for removal that depend on finalized canonical chain.
    pub(crate) fn finalize_canonical_blocks(
332     &mut self,
        finalized_block: BlockNumber,
334     num_of_additional_canonical_hashes_to_retain: u64,
    ) -> BTreeSet<BlockChainId> {
336     // get finalized chains. blocks between [self.last_finalized,finalized_block).
        // Dont remove finalized_block, as sidechain can point to it.
338     let finalized_blocks: Vec<BlockHash> = self
            .canonical_chain
340         .iter()
            .filter(|(number, _)| *number >= self.last_finalized_block && *number < finalized_block) // @audit never true
342         .map(|(_, hash)| hash)
            .collect();
344
        // remove unneeded canonical hashes.
346     let remove_until =
            finalized_block.saturating_sub(num_of_additional_canonical_hashes_to_retain);
348     self.canonical_chain.retain(|&number, _| number >= remove_until);

350     let mut lose_chains = BTreeSet::new();

352     for block_hash in finalized_blocks.into_iter() {
            // there is a fork block.
354         if let Some(fork_blocks) = self.fork_to_child.remove(&block_hash) {
                lose_chains = fork_blocks.into_iter().fold(lose_chains, |mut fold, fork_child| {
356                 if let Some(lose_chain) = self.blocks_to_chain.remove(&fork_child) {
                        fold.insert(lose_chain);
358                 }
                    fold
360             });
            }
362     }

364     // set last finalized block.
        self.last_finalized_block = finalized_block;
366
        lose_chains
368 }
```

  - The predicate at line [**341**] never holds true, so the `finalized_blocks` vector is empty.
  - Similarly, everything is retained at line [**348**] because `remove_until` is less than anything currently in the canonical chain. This does not expand the canonical chain.
  - At line [**352**] there is nothing to iterate through, so `lose_chains` is also empty.
  - Finally, `self.last_finalized_block` is set at line [**365**].

  As such, the function returns without panicking, but `self.canonical_chain` no longer contains a full `num_of_additional_canonical_hashes_to_retain` prior to `self.last_finalized_block`. Indeed, if the period of non-finality is long enough, the entry corresponding to `self.last_finalized_block` can now be missing from `self.canonical_chain`. This occurs when `new_finalized_block < old_canonical_chain.first_entry`.

  This breaks an apparent invariant where `self.canonical_chain` is expected to always hold all canonical, non-finalised blocks.

- The engine RPC returns without error, but now `BlockIndices` is in an unexpected state.

Future blocks inserted by the RPC `engine_newPayload` may incorrectly buffer blocks as disconnected, that should be attached as side-chains.

It appears that block execution is still functional despite possible gaps in the `BlockIndices::canonical_chain`. No exploits or consensus splits were identified in a subsequent call to `try_append_canonical_chain()`, as the underlying state provider passed to rEVM falls back to database access if entries cannot be found in the BlockIndices.

Scenario B: Start after some downtime or a reorg

Reth was offline for a longer period or the `headBlockHash` was in a side-chain that was not persisted to the database. Because only canonical blocks are persisted to the database (to be loaded on startup), these both mean that the received `headBlockHash` is not currently present in the `BlockchainTree`.

- From the perspective of the `BlockchainTree`, `make_canonical` is first executed. This returns a `CanonicalError::BlockchainTree(BlockchainTreeError::BlockHashNotFoundInChain { .. }))` at line [**957**].
- This is next passed through to `BeaconConsensusEngine::on_failed_canonical_forkchoice_update()`
- Then starts executing syncing via the pipeline or sync controller.
- Note that the `BlockIndices::last_finalized_block` is still the initialised value, such that it could be possible for the new target block to be part of a reorg deeper than the `last_finalized_block`.

Recommendations

Save finalised block numbers to persistent storage (e.g. to the database). On startup, load this as `last_finalized_block_number` when creating a new `BlockchainTree` and its contained `BlockIndices`.

Remove or refactor the `BlockchainTreeConfig::max_reorg_depth` field and ensure the `last_canonical_hashes` appropriately contains the finalised block and sufficient additional blocks.

Examine the codebase for other assumptions involving fixed limits to finality depth.

Evaluate whether `BlockIndices::finalize_canonical_blocks()` should still be able to handle a `finalized_block` argument that is less than `self.last_finalized_block`. If so, line [**348**] (shown below) should be modified to also *expand* `self.canonical_chain` when needed.

```
348 self.canonical_chain.retain(|&number, _| number >= remove_until);
```

Introduce appropriate testing and simulations to exercise scenarios involving long periods of non-finality and deep reorgs. Consider introducing debug assertions for invariants like that the `BlockIndices::canonical_chain` should always hold the finalized block. This can expressed in code as

```
impl BlockIndices {
  // ...
  debug_assert!(self.canonical_chain.canonical_hash(&self.last_finalized_block).is_some());
  // or
  debug_assert!(self
      .canonical_chain
      .inner()
      .first_entry()
      .map(|(&number, _)| number
          <= self
              .last_finalized_block
              .saturating_sub(num_of_additional_canonical_hashes_to_retain))
      .unwrap());
```

Resolution

The development team have mitigated the issue in PR #8473, the finalised block is updated on initialisation.

RETH-20	`expect()` Used Without Guarantees Of Success		
Asset	`reth/crates/blockchain-tree/src/blockchain_tree.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

There are 10 occurrences of `expect()` in the file `blockchain_tree.rs` (excluding tests). Some of these `expect()` statements may be reachable if other code has bugs which allow invalid conditions.

Expects will cause panics if they are triggered. It is desirable to avoid panics where possible and instead return error messages to facilitate debugging.

The following functions contain `expect()` statements:

- `post_state_data()`
- `find_all_dependent_chains()`
- `insert_unwound_chain()`
- `is_block_inside_chain()`
- `make_canonical()`

Recommendations

It is recommended to instead return an error if an `Option` is `None` or `Result` is `Err`.

Resolution

Each potential panic has been removed in PR #8278.

RETH-21	Imported Transactions May Be Removed From Fetcher Without Being Added To The Pool		
Asset	`reth/crates/net/network/src/transactions/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The transaction importer may remove items from the fetcher list without adding these transactions.

In the case where a `peer_id` is not found in the list `peers`, the transaction manager will remove the hashes from the transaction fetcher without adding them to the transaction pool.

```
fn import_transactions(
    &mut self,
    peer_id: PeerId,
    transactions: PooledTransactions,
    source: TransactionSource,
) {
    // If the node is pipeline syncing, ignore transactions
    if self.network.is_initially_syncing() {
        return
    }
    if self.network.tx_gossip_disabled() {
        return
    }

    let mut transactions = transactions.0;

    // mark the transactions as received
    self.transaction_fetcher
        .remove_hashes_from_transaction_fetcher(transactions.iter().map(|tx| *tx.hash())); // @audit transactions removed from
            ↪  fetched

    let Some(peer) = self.peers.get_mut(&peer_id) else { return }; // @audit returns without adding transactions to pool

    // ... snipped (handles the fetched transactions)
```

While it is unlikely to receive a transaction list from a `peer_id` that is not in `peers`, if this case occurs the transactions will be removed from the fetcher without adding them to `pool`.

Recommendations

It is recommended to change the order of operations such that it will first check to see if a peer exists and return if not. Then remove the transactions from the fetcher and add them to the pool.

Resolution

The order of operations has been updated to reflect the recommendation in PR #7998.

RETH-22	Incorrect Expiration Used For Checking Expired Requests		
Asset	`reth/crates/net/discv4/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The expiry calculations use the expiry for PING messages. It would be better to use the expiry associated with each request.

```
self.pending_enr_requests.retain(|_node_id, enr_request| {
    now.duration_since(enr_request.sent_at) < self.config.ping_expiration // @audit better to use enr_expiration
});

let mut failed_pings = Vec::new();
self.pending_pings.retain(|node_id, ping_request| {
    if now.duration_since(ping_request.sent_at) > self.config.ping_expiration {
        failed_pings.push(*node_id);
        return false
    }
    true
});

debug!(target: "discv4", num=%failed_pings.len(), "evicting nodes due to failed pong");

// remove nodes that failed to pong
for node_id in failed_pings {
    self.remove_node(node_id);
}

let mut failed_lookups = Vec::new();
self.pending_lookup.retain(|node_id, (lookup_sent_at, _)| {
    if now.duration_since(*lookup_sent_at) > self.config.ping_expiration { // @audit better to use neighbours_expiration
        failed_lookups.push(*node_id);
        return false
    }
    true
});
```

Recommendations

Use `self.config.enr_expiration` for pending ENRs and `self.config.neighbours_expiration` for pending lookups.

Resolution

The expirations have been updated in PRs #7507 and #7996.

RETH-23	Missing Fields Should Be Skipped When Decoding Messages		
Asset	`reth/crates/net/discv4/src/proto.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The Discv4 specification states that after EIP-8, additional list elements should be skipped when decoding packet-data.

The following types in Discv4 do not allow for additional list elements to be skipped during decoding:

- `FindNode`
- `Neighbours`
- `EnrRequest`

Similarly, for RPLx specification, addition list elements in handshake messages and Hello should be skipped. This is not enforced in the Reth RPLx implementation.

Recommendations

It is recommended to skip additional list elements to match the specifications.

Resolution

Additional fields are now skipped for RLP encoding of the listed types. Changes are reflected in PR #8039.

The issue is left open as RLPx message should also be updated to skip additional fields.

RETH-24	Unbounded Channels		
Asset	`reth/crates/consensus/beacon/src/engine/mod.rs`, `reth/crates/tokio-util/src/event_listeners.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

There is an unbounded channel for consensus events sent from the RPC.

The channel `to_engine` is used by `BeaconConsensusEngineHandle` to handle messages from the RPC. The channel is unbounded and therefore susceptible to Denial-of-Service (DoS) attacks or excessive resource consumption in times of congestion.

The RPC is only callable from the consensus client which is considered a trusted actor and therefore the likelihood of exploitation is low. However, during times of congestion, numerous events could build up. Processing events includes executing blocks and updating the blockchain tree. Thus, it may be possible that events are added to the channel faster than they may be processed, and the channel will increase in volume.

```
pub fn new(
    client: Client,
    pipeline: Pipeline<DB>,
    blockchain: BT,
    task_spawner: Box<dyn TaskSpawner>,
    sync_state_updater: Box<dyn NetworkSyncUpdater>,
    max_block: Option<BlockNumber>,
    run_pipeline_continuously: bool,
    payload_builder: PayloadBuilderHandle<EngineT>,
    target: Option<B256>,
    pipeline_run_threshold: u64,
    hooks: EngineHooks,
) -> RethResult<(Self, BeaconConsensusEngineHandle<EngineT>)> {
    let (to_engine, rx) = mpsc::unbounded_channel(); // @audit unbounded channel
```

Another unbounded channel exists in the `tokio-util` crate containing a single structure called `EventListeners`.

`EventListeners` implements a multi-producer/multi-consumer queue where each sent value is seen by all consumers. To achieve this, `EventListeners` allocates a `std::Vec` to be filled with `tokio::sync::UnboundedSender` every time `EventListeners::new_listener()` is called.

As every value sent via `EventListeners` is cloned to each `UnboundedReceiver`, and the channels are unbounded, this is prone to unlimited memory growth and eventual Out-of-Memory (OOM) conditions.

```
/// Add a new event listener.
pub fn new_listener(&mut self) -> UnboundedReceiverStream<T> {
    let (sender, receiver) = mpsc::unbounded_channel(); // @audit unbounded broadcast channel
    self.listeners.push(sender);
    UnboundedReceiverStream::new(receiver)
}
```

Note there are a number of other unbounded channels that are marked as known issues by the development team, and are therefore not included in this report.

Recommendations

It is recommended to bound all channels.

Resolution

Discussions about bounding consensus channels occur on PR #8251. It was eventually closed in favour of PR #8203 which will prune consensus messages when breaching limits.

PR #8193 sets bounded channels for event listeners.

RETH-25	Connection Denial-of-Service Condition Via Invalid TCP Packets		
Asset	`reth/crates/net/ecies/src/algorithm.rs`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Predicting a TCP sequence number of a live connection allows an attacker to close the connection by sending a datagram with the next sequence number and with garbage body.

When a datagram is received and the body message is random bytes, it will attempt to be decoded as a header. The check against the MAC will error without knowledge of the private key. Upon error, the `tokio` stream will return `Poll::Read(None)` which will eventually lead to a closed connection.

It is feasible although non-trivial to inject spoofed TCP packets into a connection if the TCP sequence number can be predicted. To predict TCP packets, a malicious user would need to have read access to the connection which may occur on local area networks.

If TCP datagrams are received with invalid body, closing the connection is common practice. However, in the case of RLPx the connection may not be re-established.

Recommendations

Recovering a TCP connection when the sequence number is known by an attacker is not feasible at the application level. The attacker will be able to send an unbounded number of TCP datagrams to increase the sequence number. To resolve the issue, a new connection must be established without knowledge of the sequence number being leaked.

RETH-26	Out-of-Bounds Access When Reading From Nippy Jar Archives		
Asset	`crates/reth/storage/nippy-jar/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

In `DataReader::offset_at`, data is read from an arbitrary index into the memory-mapped offsets file without bounds checking. As such, any index greater than the length of `offset_mmap` will cause an out-of-bounds access.

Recommendations

Add an explicit bounds check prior to performing the read from `self.offset_mmap`. Note that, the upstream dependency for this (`memmap2`) will default to the length of the file for file-backed maps.

Resolution

The recommendation has been implemented in commit 4693f73.

RETH-27	Denial-of-Service Condition Through UDP Spamming		
Asset	`reth/crates/net/discv4/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

It is possible to spam general UDP packets to block the `poll()` function from executing messages.

The `receive_loop()` will read UDP messages from the socket and send them to `poll()` via the ingress channel. The gist of the attack is to fill up the ingress channel with pointless messages to stall `poll()`. Subsequently, the process will not be able to process any genuine requests.

Once the ingress channel is full, the `receive_loop()` will block while it waits for the ingress channel to clear. During this time the OS will continue to load more UDP messages in the OS socket and eventually drop them, when full.

Note that it is possible to send the same UDP message repeatedly, which will be forwarded through the ingress channel.

The impact of filling up the ingress channel with bogus UDP messages is that it prevents valid messages from being processed.

The ingress loop has a buffer `let (ingress_tx, ingress_rx) = mpsc::channel(config.udp_ingress_message_buffer);`. Once this channel is full then `receive_loop()` will block trying to write to the ingress channel. The UDP socket will fill up and start discarding incoming packets until the `receive_loop()` can read from the OS socket.

The likelihood of the attack is low as it would require an attacker to have very large bandwidth, enough to fill the buffer before it may be processed.

Recommendations

These style of DoS attack vectors are non-trivial to fix in UDP as it is possible to forge the `from` IP address in UDP datagrams.

One consideration is to use a rate limiter which slows the number of incoming messages per IP address. By filtering out messages from the same IP before adding them to the ingress loop it would reduce the amount of spam messages without blocking the ingress loop.

However, since `from` IP address can be forged in UDP, an attacker could use this to rate limit other users by forging the `from` address to another node and then spamming packets. It is worth noting that IP spoofing on UDP is non-trivial as many internet service providers will do source packet checking to prevent IP spoofing. Thus, this is still a viable solution but with known caveats.

Additionally, consider adding some duplicate resistance and rejecting identical messages.

Resolution

Simple IP rate limiting is included in PR #8406.

RETH-28	Potential Index Out Of Bounds In `kdf()`		
Asset	`reth/crates/net/ecies/src/algorithm.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is a bug in the `kdf()` function that will occur if `dest` is ever not 32 bytes in length. Note that this is a private function and all calling instances (`decrypt_message()` and `encrypt_message()`) have the output as 32 bytes and so this issue is not reachable.

The bug occurs since `written` will be incremented by 32 until it is equal to or larger than `dest`. However, if `dest` cannot fit all 32 bytes then the slice index will be larger than the length and a panic will occur.

As an example, consider `dest.len() = 1`, for this case the first iteration of the loop will execute, and it will index `dest[0..32]` which panics since `dest` is length 1.

```
fn kdf(secret: B256, s1: &[u8], dest: &mut [u8]) {
    // SEC/ISO/Shoup specify counter size SHOULD be equivalent
    // to size of hash output, however, it also notes that
    // the 4 bytes is okay. NIST specifies 4 bytes.
    let mut ctr = 1_u32;
    let mut written = 0_usize;
    while written < dest.len() {
        let mut hasher = Sha256::default();
        let ctrs = [(ctr >> 24) as u8, (ctr >> 16) as u8, (ctr >> 8) as u8, ctr as u8];
        hasher.update(ctrs);
        hasher.update(secret.as_slice());
        hasher.update(s1);
        let d = hasher.finalize();
        dest[written..(written + 32)].copy_from_slice(&d); // @audit will index out of bounds if dest.len() % 32 != 0
        written += 32;
        ctr += 1;
    }
}
```

Recommendations

If `written + 32 > dest.len()` then only copy the number of bytes until `dest` is full.

Resolution

The resolution is to use `concat_kdf` crate which implements the required KDF. Updates can be seen in PR #7106.

RETH-29	Arithmetic Overflow In `decrypt_message()`		
Asset	`reth/crates/net/ecies/src/algorithm.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A reachable arithmetic overflow exists in the function `decrypt_message()`.

The bug occurs in the code `encrypted.len() - 32`. The overflow is reachable when the parameter `data` has a length 67-99.

In release mode this will result in a wrapping overflow. A very large number will be passed as the index to `split_at_mut()` which will error since the `encrypted` slice is less than 32 bytes in length.

The impact is that an error will occur, and the program will continue safely. Thus, this issue is rated as low severity.

```
fn decrypt_message<'a>(&self, data: &'a mut [u8]) -> Result<&'a mut [u8], ECIESError> {
    let (auth_data, encrypted) = split_at_mut(data, 2)?;
    let (pubkey_bytes, encrypted) = split_at_mut(encrypted, 65)?;
    let public_key = PublicKey::from_slice(pubkey_bytes)?;
    let (data_iv, tag_bytes) = split_at_mut(encrypted, encrypted.len() - 32)?; // @audit overflows
```

Recommendations

It is recommended to use checked maths here and return an error for the overflow case. Alternatively, minimum length checks can be used as $2 + 65 + 32 + 16 = 115$.

Resolution

The issue has been resolved in PR #7108.

RETH-30	Arithmetic Overflow In `read_body()`		
Asset	`reth/crates/net/ecies/src/algorithm.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is an overflow in the function `read_body()` when it is called with data length less than 16.

Note that the codec does round the length up to the nearest 16 bytes. However, this overflow can occur if a node sends a body with length zero and a valid header.

```
let (body, mac_bytes) = split_at_mut(data, data.len() - 16)?; //@audit potential overflow if we have a body_size 0
```

The issue is rated as low severity as the function `split_at_mut()` will raise an error on a large input.

Recommendations

The issue may be mitigated by using checked maths for the subtraction `data.len() - 16`.

Resolution

PR #7117 resolves the issue by using checked maths.

RETH-31	Arithmetic Overflow In `new_chain_fork()`		
Asset	`reth/crates/blockchain-tree/src/chain.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A potential arithmetic overflow exists in the function `new_chain_fork()`.

The bug occurs in the code `block.number - 1`. The overflow is reachable when the parameter `block.number` is 0.

```
pub(crate) fn new_chain_fork<DB, EF>(
    &self,
    block: SealedBlockWithSenders,
    side_chain_block_hashes: BTreeMap<BlockNumber, BlockHash>,
    canonical_block_hashes: &BTreeMap<BlockNumber, BlockHash>,
    canonical_fork: ForkBlock,
    externals: &TreeExternals<DB, EF>,
    block_validation_kind: BlockValidationKind,
) -> Result<Self, InsertBlockErrorKind>
where
    DB: Database + Clone,
    EF: ExecutorFactory,
{
    let parent_number = block.number - 1; // @audit potential overflow if block.number = 0
```

Recommendations

It is recommended to use checked maths here and return an error for the overflow case.

Resolution

Checked math is added in PR #8156.

RETH-32	`read_header()` Index Out Of Bounds If Input Is Not At Least 32 Bytes		
Asset	`reth/crates/net/ecies/src/algorithm.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

If the function `read_header()` is called with less than 32 bytes, an index out of bounds panic occurs.

The data is split at index 16 with the first 16 bytes being passed into `header_bytes` and the remaining bytes passed into `mac_bytes`. Following this, `mac_bytes` is accessed at index 15.

For example if `data.len() = 20`, then `mac_bytes.len() = 4` and `mac_bytes[..16]` will panic.

```
pub fn read_header(&mut self, data: &mut [u8]) -> Result<usize, ECIESError> {
    let (header_bytes, mac_bytes) = split_at_mut(data, 16)?;
    let header = HeaderBytes::from_mut_slice(header_bytes);
    let mac = B128::from_slice(&mac_bytes[..16]); //@audit will panic if data is less than 32 bytes
```

Recommendations

Use `mac_bytes.split_at(16)` to ensure it is at least length 16.

Alternatively, an initial check can be performed to ensure `data.len() >= 32`.

Resolution

The issue has been resolved in PR #7118.

RETH-33	RLP Header Is Not Validated In `RequestPair` Decoding		
Asset	`reth/crates/net/eth-wire/src/types/message.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When decoding an RLP Header for `RequestPair`, the header is decoded then discarded.

The `decode()` function will first decode an RLP header and then the `request_id` and finally the message. However, the header is not validated for `header.payload_length`, therefore it is possible to read bytes past the end of the RLP encoding.

```
impl<T> Decodable for RequestPair<T>
where
    T: Decodable,
{
    fn decode(buf: &mut &[u8]) -> alloy_rlp::Result<Self> {
        let _header = Header::decode(buf)?; //@audit we discard the header without checking payload_length
        Ok(Self { request_id: u64::decode(buf)?, message: T::decode(buf)? })
    }
}
```

The impact of this issue is low, as the program will decode the bytes and adjust the buffer as required. However, for correctness we should ensure we do not read past the `payload_length` when decoding the message and request ID.

Recommendations

Check the length of the bytes read by `u64::decode()` and `T::decode()` and ensure the amount read is less than `payload_length`.

Resolution

The recommendation has been implemented in PR #7292 by validating the length of bytes consumed matches the RLP header.

RETH-34	RLP Header Is Not Validated In `DisconnectReason`		
Asset	`reth/crates/net/eth-wire/src/disconnect.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A `DisconnectReason` may be decoded with a RLP header that contains an empty list and empty payload, that is `header.payload_length = 0` after `header = Header::decode(buf)`.

The `decode()` function will continue to read the next byte of the buffer which is outside the payload since the `payload_length = 0`.

```
if buf.len() > 1 {
    // this should be a list, so decode the list header. this should advance the buffer so
    // buf[0] is the first (and only) element of the list.
    let header = Header::decode(buf)?;  //@audit it is possible to have payload_length = 0
    if !header.list {
        return Err(RlpError::UnexpectedString)
    }
}

// geth rlp encodes [`DisconnectReason::DisconnectRequested`] as 0x00 and not as empty
// string 0x80
if buf[0] == 0x00 {
    buf.advance(1);
    Ok(DisconnectReason::DisconnectRequested)
} else {
    DisconnectReason::try_from(u8::decode(buf)?)
        .map_err(|_| RlpError::Custom("unknown disconnect reason"))
}
```

It does not cause an index out of bounds or any other issues, however it is considered invalid to read the next byte of the buffer if the `payload_length` is zero.

Recommendations

Consider adding a check to ensure `payload_length = 1`.

Resolution

The recommendation is implemented in PR #7284.

RETH-35	Index Out Of Bounds Panic When Sending Empty Bytes		
Asset	`reth/crates/net/eth-wire/src/p2pstream.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

An index out of bounds (IOOB) panic will occur when empty bytes are sent as a P2P message.

```
fn start_send(self: Pin<&mut Self>, item: Bytes) -> Result<(), Self::Error> {
    if item.len() > MAX_PAYLOAD_SIZE {
        return Err(P2PStreamError::MessageTooBig {
            message_size: item.len(),
            max_size: MAX_PAYLOAD_SIZE,
        })
    }

    // ensure we have free capacity
    if !self.has_outgoing_capacity() {
        return Err(P2PStreamError::SendBufferFull)
    }

    let this = self.project();

    let mut compressed = BytesMut::zeroed(1 + snap::raw::max_compress_len(item.len() - 1)); // @audit item.len() - 1 will overflow
        ↪  if empty
    let compressed_size =
        this.encoder.compress(&item[1..], &mut compressed[1..]).map_err(|err| { //@audit index out of bounds panic if item.len() =
            ↪  0
```

If `start_send()` is called with `item.len() = 0` then there will be an overflow in `item.len() - 1`. Furthermore, this will cause an index out of bounds panic in `item[1..]`.

It is not expected to be able to send data of length zero, hence the likelihood of this issue is rated as low.

Recommendations

Return an error if `item.len() = 0`.

Resolution

Each `item` is checked to be non-empty in PR #7294.

RETH-36	Index Out Of Bounds Panic When Multiplex Message Is Empty		
Asset	`reth/crates/net/eth-wire/src/multiplex.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

An index out of bounds (IOOB) panic will occur when empty bytes are received as a multiplex message from the P2P stream.

The following snippet is taken from `into_satellite_stream_with_tuple_handshake()`.

```
tokio::select! {
    Some(Ok(msg)) = self.inner.conn.next() => {
        // Ensure the message belongs to the primary protocol
        let offset = msg[0]; //@audit potential index out of bounds
        if let Some(cap) = self.shared_capabilities().find_by_relative_offset(offset).cloned() {
            // ... snipped
    }
    Some(msg) = from_primary.recv() => {
        self.inner.conn.send(msg).await.map_err(Into::into)?;
    }
    // ... snipped
}
```

Similarly, the function `poll_next()` may also cause an index out of bounds panic on an empty incoming message.

```
loop {
    // pull messages from connection
    match this.inner.conn.poll_next_unpin(cx) {
        Poll::Ready(Some(Ok(msg))) => {
            delegated = true;
            let offset = msg[0]; // @audit potential index out of bounds panic
```

There are two potential index out of bounds panics which both occur when the P2P message received is empty. However, there are checks in `p2pstream.rs::poll_next()` to ensure that incoming bytes are non-zero. Thus, the likelihood of exploitation for this issue is deemed low.

Recommendations

It is recommended to replace `msg[0]` with `fuzzing` and propagate any errors for each of `poll_next()` and `into_satellite_stream_with_tuple_handshake()`, rather than indexing a potentially empty slice.

Resolution

The recommendation was implemented in PR #7314.

RETH-37	Arithmetic Overflows In `ProtocolStream` & `ProtocolProxy`		
Asset	`reth/crates/net/eth-wire/src/multiplex.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There are overflows in the addition and subtraction of the relative message ID offset in `ProtocolStream` and `ProtocolProxy`. Additionally, each of the functions will panic if the `msg.len() = 0`.

One example is `mask_msg_id()` in `ProtocolProxy`. If `msg.len() = 0` there will be an index out of bounds panic in `msg[0]`. Additionally, `msg[0] + self.shared_cap.relative_message_id_offset()` may overflow if `msg[0]` is large.

```
fn mask_msg_id(&self, msg: Bytes) -> Bytes {
    let mut masked_bytes = BytesMut::zeroed(msg.len());
    masked_bytes[0] = msg[0] + self.shared_cap.relative_message_id_offset(); // @audit index out of bounds panic and overflow
    masked_bytes[1..].copy_from_slice(&msg[1..]);
    masked_bytes.freeze()
}
```

The following functions are vulnerable to the same issues:

- `ProtocolProxy::mask_msg_id()`
- `ProtocolProxy::unmask_id()`
- `ProtocolStream::mask_msg_id()`
- `ProtocolStream::unmask_id()`

Recommendations

It is recommended to return an error when insufficient data is passed as input. Furthermore, use checked maths and handle the overflow case.

Resolution

Overflow checks are included in PR #7297.

RETH-38	Forks With `next` As Timestamp May Be Confused With Blocknumber		
Asset	`reth/crates/ethereum-forks/src/forkid.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

There is insufficient differentiation between whether the field `next` is used to represent a timestamp vs a blocknumber.

```
    // We check if this fork is time-based or block number-based
    // NOTE: This is a bit hacky but I'm unsure how else we can figure out when to use
    // timestamp vs when to use block number..
    let head_block_or_time = match self.cache.epoch_start {
        ForkFilterKey::Block(_) => self.head.number,
        ForkFilterKey::Time(_) => self.head.timestamp,
    };

    //... compare local head to FORK_NEXT.
    return if head_block_or_time >= fork_id.next { //@audit if next is a timestamp and head is a block we'll never pass
        // 1a) A remotely announced but remotely not passed block is already passed locally,
        // disconnect, since the chains are incompatible.
        Err(ValidationError::LocalIncompatibleOrStale {
            local: self.current(),
            remote: fork_id,
        })
    } else {
        // 1b) Remotely announced fork not yet passed locally, connect.
        Ok(())
    }
}
```

The issue will occur if `self.cache.epoch_start` is measured in blocknumbers and `fork_id.next` is measured in timestamps.

The field is used to determine whether a connection should be established with the peer.

Since block numbers are significantly less than timestamps, if the timestamp is somewhat recent it will always be larger. The current mainnet block height of Ethereum is about 19 million whereas the current timestamp for Unix is almost 2 billion.

Recommendations

A possible solution to deciding whether to use a timestamp or header could be to check if `fork_id.next` is greater than an old timestamp. e.g. the timestamp of 2011 before Ethereum mainnet was 1.3 billion.

Consider estimating if `fork_id.next > 1,300,000,000` then it is expected to be a timestamp and if it is less, then the units are in blocknumbers.

An extra safety check could be added that the estimation does not occur if `self.head.number > 1,300,000,000` to future-proof for when Ethereum has over a billion blocks.

Resolution

The recommendation has been implemented in PR #8320.

RETH-39	`PING/PONG` Man-in-the-Middle
Asset	`reth/crates/net/discv4/src/lib.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

There is a lack of checks for the fields `Ping.to`, `Ping.from`, `Pong.to`; instead, the sender IP address is recorded. Since PING and PONG messages are replayable until the expiry timestamp, this opens up an attack vector where a user may MitM (Man-in-the-Middle) the PING-PONG steps.

An example of the issue is to have a malicious user Mallory attack Alice and Bob.

Steps:

1. Mallory waits for a PING from Alice. Mallory copies the exact message and forwards the PING to Bob.
2. Bob registers Mallory's IP address and UDP port in the `NodeRecord` along with Alice's ID and sends both:
   (a) PONG back to Mallory
   (b) PING to Mallory
3. Mallory will forward both PING and PONG to Alice
4. Alice will receive two messages:
   (a) PONG - runs `on_pong()` which contains the correct echo hash and will be processed
   (b) PING - will return a PONG back to Mallory
5. Mallory forwards the PONG message to Bob

The result here is the Bob will have registered Mallory's IP address with Alice's ID.

It is possible to continue this MitM attack with ENR requests also to establish an endpoint proof.

Geth is also vulnerable to the same attack as seen in v4_udp.go.

Recommendations

Consider adding two steps to resolve the issue:

- Verify that the `to` address matches the local configuration for both PING and PONG messages;
- Ensure the remote address matches the `Ping.from`.

However, it is worth investigating more before implementing this solution, as there may be a good reason Geth has implemented it that way (potentially to facilitate proxies).

Resolution

The issue is marked as won't fix by the development team.

RETH-40	Database Shrinking Accidentally Enabled
Asset	`crates/storage/db/src/implementation/mdbx/mod.rs` & `crates/storage/libmdbx-rs/src/environment.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Reth configures libmdbx to allow database shrinking but the intent is to disable database shrinking.

Reth sets `libmdbx` configuration in `DatabaseEnv::open`, with the relevant excerpt displayed below.

```
233 inner_env.set_geometry(Geometry {
        // Maximum database size of 4 terabytes
235     size: Some(0..(4 * TERABYTE)),
        // We grow the database in increments of 4 gigabytes
237     growth_step: Some(4 * GIGABYTE as isize),
        // The database never shrinks
239     shrink_threshold: None,
        page_size: Some(PageSize::Set(default_page_size())),
241 });
```

The `eth_newPayloadV3()` field is set to `None` and appears, at first glance, to have the intended effect of disabling database shrinking. However, the `None` value actually results in the default of −1 being passed through to the underlying implementation at `crates/storage/libmdbx-rs/src/environment.rs:643` (shown below).

```
mdbx_result(ffi::mdbx_env_set_geometry(
    // ... snipped
    geometry.shrink_threshold.unwrap_or(-1),
    // ... snipped
))?;
```

According to the libmdbx API documentation

> ***shrink_threshold*** *"The shrink threshold in bytes, must be greater than zero to allow the database to shrink and greater than growth_step to avoid shrinking right after grow. Negative value means "keep current or use default". Default is 2*growth_step."*

Hence, the resulting value for `shrink_threshold` is actually 8 GiB.

The testing team did not identify a security risk associated with this issue, hence an *informational* severity.

Recommendations

Confirm whether database shrinking is intended. If disabling database shrinking is desired, instead pass the following value to `NodeRecord`

```
crates/storage/libmdbx-rs/src/environment.rs:643
```

Resolution

The issue is resolved in PR #8324.

RETH-41	Database Not Opened in Exclusive Mode
Asset	`crates/storage/db/src/implementation/mdbx/mod.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The Reth database `libmdbx` supports an exclusive mode[3] to prevent the database from being opened by multiple processes at once. As Reth runs as a single process, there is no need or expectation that the database is accessed by multiple processes. This feature is not enabled and could result in accidental database corruption if multiple instances of Reth were started.

This feature is configured in Reth via the `DatabaseArguments::exclusive` struct field of type `Option<bool>`. The associated doc comment states that "If `None`, the default value is used" but does not clarify what that default value is. At line [284] we observe that the value passed to the inner implementation defaults to the `bool::default()` value of `false`.

```
exclusive: args.exclusive.unwrap_or_default(),
```

Currently this default is always used by Reth binaries and is not exposed for configuration by the user.

The identified risks are associated with misuse rather than security, hence an *informational* severity.

Recommendations

Consider enabling exclusive mode for Reth, and enabling it by default. This could be achieved without modifying the forked `reth-libmdbx` crate by setting the flag at line [284] to `true` by default.

```
decode_enveloped()
```

If setting it to exclusive by default, ensure this is appropriately documented.

Resolution

The development team have stated the intention is for the database to be opened in non-exclusive mode.

[3] Documented here and in the doc comments for `DatabaseArguments::exclusive` at line [72].

RETH-42	ENR Responses Are Not Validated
Asset	`reth/crates/net/discv4/src/lib.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When an ENR Response packet is received in the function `on_enr_response()`, the value of `msg.enr` is not validated then immediately discarded.

The Discv4 specification states the following:

> *"The recipient of the packet should verify that the node record is signed by the public key which signed the response packet."*

Note that this is the same behaviour in Geth excluding a small amount of verification of the ENR record. This can be seen in v4_udp.go#L361-L375 and v4_udp.go#L782.

Recommendations

Consider performing validation on the incoming ENR responses.

Resolution

Validation has been added to ensure the record has been signed by the public key in PR #8407.

RETH-43	Eclipse Mitigations
Asset	`reth/crates/net/discv4/*`
Status	**Open**
Rating	Informational

Description

One the main attacks against any discovery protocol is an eclipse. An attacker generates a large and occasionally specific set of node-ids in order to populate the first 10 or so buckets routing table of a node with malicious entries. Once achieved, the attacker can force all new peer discoveries to be malicious peers of their choosing.

In practice, an attacking discv4 node pre-generates or generates on the fly many node-ids and on a query request, responds with these malicious nodes, which can often relate back to their attacking node. Over time, they end up filling the local routing table by removing previous honest entries.

Recommendations

There are a few ways to mitigate this: one mitigation is to place limits on the kinds of entries we store in our routing table. There are two kinds of limits used in Lighthouse, one is an IP limit per bucket. There is a limit of IPs in subnets as unlike node-ids, IPs are harder to generate. A write-up on this issue exists in the ethereum/devp2p Github repository.

RETH-44	ECIES Protocol Bugs
Asset	`reth/crates/net/ecies/*`
Status	**Open**
Rating	Informational

Description

There are bugs within the ECIES protocol that are not specific to the Reth implementation. While these are security considerations, the security impact is not significant enough to warrant immediate patching.

Forgeable signatures

The function `recover_ecdsa()` allows recovering a public key from a message hash and signature. It is trivial to forge a valid signature if the message hash is selected by the attacker. In ECIES the attacker is able to set the message hash to `x ^ nonce` where `x` is the ECDH x-coordinate and `nonce` is arbitrarily chosen by the attacker.

An attacker is able to select an arbitrary message hash. To set an arbitrary message hash, first calculating ECDH `x` then select the desired final message hash. Finally, set `nonce = x ^ messageHash`, such that `messageHash = x ^ nonce`.

The impact here is the attacker can choose any `remote_ephemeral_public_key` without knowing the secret key. The impact is not severe, as this ephemeral key is only used to calculate the shared secret. If the attacker does not know the shared secret they cannot encrypt or decrypt messages with this peer. The connection will not be able to share `Hello` messages and will error or be dropped after a timeout.

Authentication without knowledge of public key secret

It is possible to pass the ECIES Auth / Ack handshake without knowledge of a public key secret. This also occurs due to `recover_ecdsa()`, in that if we provide a random message and signature there is around 50% chance it will succeed and return a valid public key, to which no one knows the secret for.

Again this style of attack will result in an encrypted connection to which the attacker does not know the shared secret. This will allow them to open a connection but unable to encrypt and share the `Hello` message resulting in an error or timeout.

Auth / Ack packets are replayable

There is no expiry on Auth or Ack packets. Therefore, an attacker is able to re-use an existing handshake by replaying an Auth packet. This would result in a handshake without knowledge of the shared secret, and as such, no more messages can be encrypted or decrypted by the attacker.

Recommendations

Forgeable signatures

The `x` and `nonce` values should be concatenated then hashed rather than using XOR. This would prevent the attacker from selecting a specific message hash.

Authentication without knowledge of public key secret

To resolve this issue, include the public key in the message hash. That is hash `ephemeral_public_key`, `x` and `nonce`. To facilitate the message hashing the ephemeral public key would need to be passed as a field in the message.

Auth / Ack packets are replayable

Auth and Ack messages should have an expiry timestamp or nonce. A timestamp is the quickest and easiest way to prevent replay though it allows replay before the timestamp expires. A nonce is more secure but requires long term storage which will have resource consumption considerations.

RETH-45	Large `base_fee` Overflows Block Base Fee Calculations
Asset	`crates/reth/primitives/src/basefee.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

A large `base_fee` may cause an overflow when adding change. This would require the previous base fee plus the increase to overflow a `u64`, which is unlikely.

```
pub fn calculate_next_block_base_fee(
    gas_used: u64,
    gas_limit: u64,
    base_fee: u64,
    base_fee_params: crate::BaseFeeParams,
) -> u64 {
    // Calculate the target gas by dividing the gas limit by the elasticity multiplier.
    let gas_target = gas_limit / base_fee_params.elasticity_multiplier;


    match gas_used.cmp(&gas_target) {
        // If the gas used in the current block is equal to the gas target, the base fee remains the
        // same (no increase).
        std::cmp::Ordering::Equal => base_fee,
        // If the gas used in the current block is greater than the gas target, calculate a new
        // increased base fee.
        std::cmp::Ordering::Greater => {
            // Calculate the increase in base fee based on the formula defined by EIP-1559.
            base_fee + // @audit can overflow if base_fee is large
                (std::cmp::max(
                    // Ensure a minimum increase of 1.
                    1,
                    base_fee as u128 * (gas_used - gas_target) as u128 /
                        (gas_target as u128 * base_fee_params.max_change_denominator as u128),
                ) as u64)
        }
```

Recommendations

Consider using `saturating_add()`.

Resolution

The issue is marked as won't fix by the development team. The variable `base_fee` has been increase from a `u64` to `u128` and is therefore unlikely to overflow.

RETH-46	Missing Documentation for Untrusted `NippyJar` and `Compact` Formatted Data
Asset	`crates/storage/codecs` & `crates/storage/nippy-jar`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `NippyJar` and `Compact` encoding formats and their implementations are designed for storing and retrieving data internally. They are not hardened to safely read potentially malicious data. Documentation should clearly and visibly warn against misuse.

For example, the `Compact` encoding does not allow limiting the length values to protect against allocating extremely large vectors. The implementation can trivially panic after reading a length value that is larger than the size of the buffer being read from (out of bounds with a range slicing operation). Similarly, the `decode_varuint()` function will panic with `"could not decode varuint"` when passed malformed data.

The `NippyJar` implementation can similarly panic after reading offset values that exceed the bounds of the data file.

The testing team notes that these formats are used safely in Reth for internal storage purposes. However, because these modular components are also intended to be used as libraries in other projects, it is important that documentation warns against their misuse.

Recommendations

Ensure crate documentation and `README` files clearly warn against using the `Compact` and `NippyJar` formats to read untrusted data.

Resolution

Documentation has been added in PR #8345.

RETH-47	Missing Panic Comments in `from_compact()`
Asset	`crates/reth/primitives/src/transaction/mod.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Numerous implementations of the trait `Compact`, specifically the function `from_compact()`, may panic.

There are `unreachable!()` statements that would cause a panic if reached. These panics could be triggered on certain input. However, the calling function is the compact codec in storage and so these will only be reached if bad data is added to the storage database. As these panics are not reachable unless there is an invalid database entry, this is not likely to occur. However, there should be doc comments stating how and why these panics could occur.

For example, `Transaction::from_compact()` has multiple `unreachable!()` statements that could be reached with an identifier larger than 3.

```
fn from_compact(mut buf: &[u8], identifier: usize) -> (Self, &[u8]) {
    match identifier {
        0 => {
            let (tx, buf) = TxLegacy::from_compact(buf, buf.len());
            (Transaction::Legacy(tx), buf)
        }
        1 => {
            let (tx, buf) = TxEip2930::from_compact(buf, buf.len());
            (Transaction::Eip2930(tx), buf)
        }
        2 => {
            let (tx, buf) = TxEip1559::from_compact(buf, buf.len());
            (Transaction::Eip1559(tx), buf)
        }
        3 => {
            // An identifier of 3 indicates that the transaction type did not fit into
            // the backwards compatible 2 bit identifier, their transaction types are
            // larger than 2 bits (eg. 4844 and Deposit Transactions). In this case,
            // we need to read the concrete transaction type from the buffer by
            // reading the full 8 bits (single byte) and match on this transaction type.
            let identifier = buf.get_u8() as usize;
            match identifier {
                3 => {
                    let (tx, buf) = TxEip4844::from_compact(buf, buf.len());
                    (Transaction::Eip4844(tx), buf)
                }
                #[cfg(feature = "optimism")]
                126 => {
                    let (tx, buf) = TxDeposit::from_compact(buf, buf.len());
                    (Transaction::Deposit(tx), buf)
                }
                _ => unreachable!("Junk data in database: unknown Transaction variant"), // @audit should have a panic comment for
                    ↪ this case
            }
        }
        _ => unreachable!("Junk data in database: unknown Transaction variant"), // @audit should have a panic comment for this
            ↪ case
    }
}
```

Recommendations

It is recommended to update the transaction signature of `from_compact()` to return an error. This error can handle any of the deserialisation issues that may occur and exit gracefully.

Resolution

Additional documentation has been added in PR #8346.

RETH-48	`is_database_empty()` False Positive For Paths That Are Not Directories
Asset	`crates/storage/db/src/utils.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The utility function `is_database_empty()` takes a path parameter and returns a boolean value indicating whether the path corresponds to an empty database. The function returns `true` for paths that exist but are not directories, which may be unexpected.

Though unusual, this could be encountered if the Reth data directory contains a file named "db".

No security implications were identified for this issue, hence an *informational* severity.

Detail

The following test function constitutes a proof of concept. The test fails if `is_database_empty()` returns `true` when passed a path to a non-empty file.

```
#[test]
fn not_empty_if_db_path_is_a_file() {
  let base_dir = tempdir().unwrap();
  let db_file = base_dir.as_ref().join("db");
  fs::write(&db_file, b"Lorem ipsum").unwrap();

  let result = is_database_empty(&db_file);
  // would expect the function to return false
  assert!(!result);
}
```

When the same path is passed to `init_db()` at `crates/storage/db/src/lib.rs:97`, the path is treated as if it is a valid but empty database directory. Fortunately an error is safely returned at line [**103**] when trying to create a directory at that path:

```
Error {
  msg: "Could not create database directory /tmp/.tmp6h66uI/db",
  source: CreateDir {
    source: Os {
      code: 17,
      kind: AlreadyExists,
      message: "File exists",
    },
    path: "/tmp/.tmp6h66uI/db",
  },
},
```

Recommendations

Consider whether `is_database_empty()` should return `false` or an error when passed a path to a non-directory.

Resolution

The recommendation has been implemented in PR #8351.

RETH-49	`BlockchainTreeConfig` Concerns Regarding Fixed Finalisation Depth
Asset	`crates/blockchain-tree/src/config.rs`
Status	Open
Rating	Informational

Description

There are several inaccuracies in the `BlockchainTreeConfig` that appear to indicate misunderstandings surrounding consensus layer's fork-choice and finality mechanisms.

Consider the following excerpt that defines the default values for `BlockchainTreeConfig`:

```
28  // Gasper allows reorgs of any length from 1 to 64.
    max_reorg_depth: 64,
30  // This default is just an assumption. Has to be greater than the `max_reorg_depth`.
    max_blocks_in_chain: 65,
```

1. The comment at line [28] is inaccurate. When the network is unhealthy and unable to finalise, it is possible to have reorgs of a depth with no fixed bounds (much greater than 64).
2. Relying on a `max_reorg_depth` is potentially dangerous. From the perspective of the Ethereum protocol, there is no "maximum reorg depth" other than the last finalised block. If the implementation's correctness relies on any fixed depth value, it may fail to agree with other EL implementations during adverse network conditions (when consensus is most important).
3. The `max_blocks_in_chain` field is unused in the codebase under review and has no effect.

This issue is focused on the comments and config definitions, which do not appear to pose a direct security risk, hence an *informational* severity rating.

Recommendations

Evaluate whether these findings indicate misunderstood assumptions that need to be rectified elsewhere in the codebase and design. Additionally:

1. Correct or remove the comment at line [28].
2. If making use of a fixed `max_reorg_depth` in the in-memory Blockchain Tree, ensure there is an alternate recovery pathway that allows processing larger reorgs. This could involve rebuilding the state from some on-disk checkpoint. In this case, consider also renaming and documenting the field to indicate that the limit is only for Blockchain Tree, rather than Reth as a whole.

   Otherwise, consider removing the `max_reorg_depth` field and modifying code that relies on it.
3. Consider whether `max_blocks_in_chain` should remain unused. If so, remove the unnecessary field.

RETH-50	Miscellaneous General Comments
Asset	`/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **`muxdemux.rs` is unused:**
   The file and associated code is not used anywhere in Reth and could be deleted. Consider removing the unused code if not otherwise useful (e.g. intended for library users).
2. **Unnecessary, potentially lossy casts:**
   At `crates/storage/nippy-jar/src/lib.rs`, `DataReader::offset_size` is stored as a `u64` type and typically used by casting it to a `usize` like `self.offset_size as usize`. This can be a lossy, potentially truncating cast on some architectures where the `usize` type is smaller than a `u64` (like 32bit x86).
   This can be avoided by instead defining `offset_size` as a `u8` type, which is sufficient to hold the value retrieved from disk.
3. **Confusing blockchain-tree function naming:**
   (a) The function `BlockChainTree::is_block_hash_inside_chain()` (defined at `crates/blockchain-tree/src/blockchain_tree.rs:226`) is a confusing and misleading name. From the implementation and doc comment, it is clear that the functionality is more aptly described with "in side-chain" rather than "inside chain" (as it first appears).
   Consider renaming to `is_block_hash_in_sidechain` or similar.
   The same struct also has a very similarly named function `is_block_inside_chain()` that returns whether a block is present in the `BlockIndices`, and has a doc comment including "inside chain".
   Evaluate whether `is_block_inside_chain` should also be renamed.
   (b) In `crates/blockchain-tree/src/block_indices.rs`, the comments for `get_canonical_block_number()` and `is_block_hash_canonical()` mention a "canonical chain", but the implementation is only checking canonical *non-finalized* blocks. This is also true for `get_canonical_block_number()` defined at `crates/blockchain-tree/src/canonical_chain.rs:49`.
   Consider renaming to reduce confusion, and ensure doc comments are clear.
   (c) Similarly, `crates/blockchain-tree/src/canonical_chain.rs` defines `get_canonical_block_number()` and `canonical_number()` functions, which are quite ambiguously named. On inspection, `get_canonical_block_number()` checks through non-finalized canonical blocks, and `canonical_number()` iterates over the whole canonical chain cached in the index.
   Consider renaming and making doc comments more clear.
4. **Validation of non-terminal difficulty blocks not implemented:**
   Noted as a `TODO`, if a consensus client sends a block before total terminal difficulty then it will not be properly validated.

This should be fine since the consensus client will send via `engine_newPayload` from the Engine API which sets difficulty to zero for new blocks.

Only an issue during syncing if malformed blocks are received.

5. **Unused functions may be removed:**
   The following functions are unused and may be removed:
   - `validate_transaction_regarding_header()`
   - `validate_all_transaction_regarding_block_and_nonces()`
6. **Chain split allows block number larger than the chain:**
   If the function `split()` is called with a block number greater than the tip, it will consider this split valid and treat it the same as splitting the canonical head.
   If `block_number > chain_tip`, then it implies the block is not in this chain and the chain cannot be split.
   Consider returning an error for this case instead of `NoSplitPending`.
7. **`reth-codecs` documentation issues:**
   (a) In `crates/storage/codecs/README.md`, the Features section is out-of-date. There is no mention of the `compact` encoding that is now the default (main codec).
   (b) There is no documentation detailing or specifying the `Compact` encoding format. The `docs/design/codecs.md` file is empty, the crate `README.md` does not mention the format, and the doc comments give little detail in terms of *how* the standard and `varuint` types are encoded.
8. **`reth-blockchain-tree` documentation issues:**
   (a) At `crates/blockchain-tree/src/blockchain_tree.rs:138`, the doc comment states that `is_block_known()` returns an error if "*the block is already finalized*". This is not accurate and it is more clear to state that an error is returned when the block is not part of the canonical chain but is at a height that is already finalised. (The block itself is not finalised.)
9. **Project structure documentation issues:**
   At `docs/repo/ci.md`, line [10] contains a broken link to a fuzz github workflow that does not currently exist.

Recommendations

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

Resolution

The development team have acknowledged these findings, addressing them where appropriate as follows:

1. Resolved in PR #8287.
2. Resolved in PR #8360.
3. Resolved in PR #8408.
4. Unresolved.
5. Resolved in PR #7972.

6. Resolved in PR #8285.
7. Resolved in PR #8665.
8. Resolved in PR #8408.
9. Resolved in PR #8363.

RETH-51	Missing Payload Header Validation For Blob Fields, Withdrawals
Asset	`crates/payload/validator/src/lib.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `ensure_well_formed_payload()` lacks checks for the fields `block.header.blob_gas_used` and `block.header.excess_blob_gas` to ensure they are `None` before the Cancun update timestamp.

For a V3 execution payload the function `try_payload_v3_to_block()` allows setting `blob_gas_used` and `excess_blob_gas` to `Some()` value. There lacks checks to ensure these values are set to `None` before Cancun.

Note similarly for Shanghai, `Withdrawals` may be added to blocks prior to Shanghai upgrade. Note that although the fork is passed if blocks are sent with a timestamp before Shanghai these should not have withdrawals.

Furthermore, `cancun_fields.parent_beacon_block_root()` should only return `Some()` after the Cancun upgrade. If the value is `Some()` it will be set in the function `try_into_block()`.

```
98    pub fn ensure_well_formed_payload(
      &self,
100   payload: ExecutionPayload,
      cancun_fields: MaybeCancunPayloadFields,
102 ) -> Result<SealedBlock, PayloadError> {
      let block_hash = payload.block_hash();
104
      // First parse the block
106   let block = try_into_block(payload, cancun_fields.parent_beacon_block_root())?; // @audit may set `parent_beacon_block_root`
          ↪ before Cancun
108   let cancun_active = self.is_cancun_active_at_timestamp(block.timestamp);
110   if !cancun_active && block.has_blob_transactions() { // @audit lacks checks for block.header excess_blob_gas and blob_gas_used
          // cancun not active but blob transactions present
112       return Err(PayloadError::PreCancunBlockWithBlobTransactions)
      }
114
      // @audit should include checks for Shanghai withdrawals
116
      // Ensure the hash included in the payload matches the block hash
118   let sealed_block = validate_block_hash(block_hash, block)?;
120   // EIP-4844 checks
      self.ensure_matching_blob_versioned_hashes(&sealed_block, &cancun_fields)?;
122
      Ok(sealed_block)
124 }
```

The severity is rated as informational severity as the end point is only callable through the authenticated EngineAPI and the consensus layer "*should*" be calling the correct version based on the incoming block timestamp.

Recommendations

Add checks to `ensure_well_formed_payload()` to ensure that these fields are `None` before Cancun and `Some()` afterwards.

Additionally, add checks for the Shanghai fork to ensure that withdrawals are not included before this timestamp and are included afterwards.

Finally, ensure `cancun_fields.fields` is `None` before Cancun and `Some()` afterwards.

Resolution

The issue was resolved in PR #7993 and alloy PR #649.

Appendix A Vulnerability Severity Classification

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

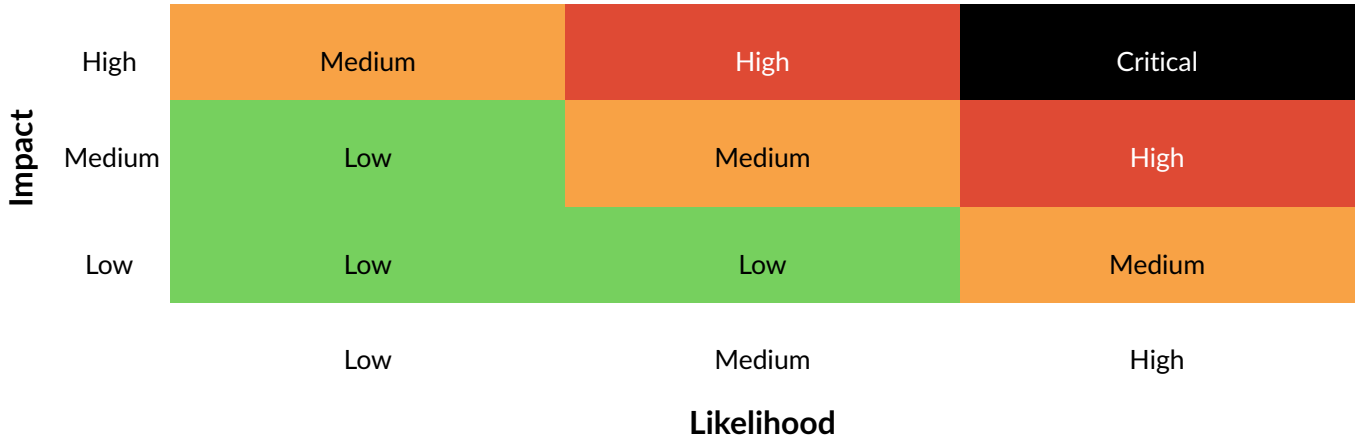

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References