Berachain

bera-reth & bera-geth

Security Assessment Report

Version: 2.0

September, 2025

Contents

Introduction

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

Disclaimer

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

Document Structure

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

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

Overview

`BRIP-0004` proposes enshrining the `distributeFor()` smart contract function call into the execution layer, enabling automatic Proof of Liquidity reward distribution at the beginning of each block without relying on external transactions or bots.

The implementation introduces a new PoL transaction type (`0x7E`) that automatically calls `distributeFor()` for the previous block's proposer at the start of each block, alongside modifications to both `bera-geth` and `bera-reth` execution clients to support enshrined system calls and enhanced forkchoice attributes containing parent proposer public keys.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the berachain/bera-reth and berachain/bera-geth repositories.

The scope of this time-boxed review was strictly limited to changes in PRs bera-reth#39 and bera-geth#21 at the commits 53363e4 and 688d070.

The fixes of the identified issues were assessed at the commits f1d940f and 2446ddec for `bera-reth` and `bera-geth` respectively.

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

Approach

The security assessment covered components written in Golang and Rust.

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

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

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

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

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

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

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

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

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

The manual review was also supplemented with fuzz testing techniques to identify potential vulnerabilities. Fuzzing involves providing randomized and unexpected data inputs to the software to uncover bugs, crashes (panics in Rust), and other unexpected behaviours.

The fuzzing process focused on identifying issues such as:

- Memory corruption and buffer overflows
- Broken invariants and state consistency
- Resource exhaustion (memory, CPU)
- Infinite or extended loops
- Unexpected crashes and panics

Output for these automated tools is available upon request.

Coverage Limitations

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

Findings Summary

The testing team identified a total of 20 issues during this assessment. Categorised by their severity:

- Critical: 4 issues.
- Medium: 1 issue.
- Low: 5 issues.
- Informational: 10 issues.

Detailed Findings

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

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

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

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

Summary of Findings

ID	Description	Severity	Status
BRG4-01	Inconsistent PoL Transaction Validation Before Prague1	Critical	Resolved
BRG4-02	PoL Gas Limit Consensus Mismatch Between Clients	Critical	Resolved
BRG4-03	`bera-reth` Missing RLP Length Validation During PoL Transaction Decoding	Critical	Resolved
BRG4-04	Prague1 Empty Block Consensus Mismatch Between Clients	Critical	Resolved
BRG4-05	`bera-reth` PoL Transaction Sender Vector Misalignment	Medium	Closed
BRG4-06	Engine API Fork Validation Discrepancy Between Clients	Low	Resolved
BRG4-07	`bera-geth` PoL Transaction Simulation Bypasses System Address Check	Low	Resolved
BRG4-08	`bera-geth` `ethclient` PoL Transaction Signature Validation Failure	Low	Resolved
BRG4-09	`bera-reth` PoL Transaction Trait Implementation Is Incorrect	Low	Resolved
BRG4-10	`bera-geth` PoL Transaction Error Mutation Breaks Error Type Checking	Low	Resolved
BRG4-11	`bera-geth` Incorrect PoL Transaction Gas Price	Informational	Resolved
BRG4-12	`bera-reth` Payload ID Is Missing `prev_proposer_pubkey`	Informational	Resolved
BRG4-13	`bera-geth` PoL Transaction JSON Marshalling Missing Case	Informational	Resolved
BRG4-14	`bera-geth` PoL Transaction JSON Marshalling Redundant Value Check	Informational	Resolved
BRG4-15	`bera-geth` PoL Transaction Should Return Error For `nil` Pubkey	Informational	Resolved
BRG4-16	`bera-geth` Engine API Fork Validation Bypass After Osaka Activation	Informational	Resolved
BRG4-17	Default Configuration Discrepancies Between Clients	Informational	Closed
BRG4-18	`bera-reth` Prague1 Fork Validation Missing Prague Check	Informational	Resolved
BRG4-19	`bera-reth` Unmaintained Dependencies	Informational	Closed
BRG4-20	Miscellaneous General Comments	Informational	Resolved

BRG4-01	Inconsistent PoL Transaction Validation Before Prague1		
Asset	`bera-reth/src/consensus/mod.rs, bera-geth/core/block_validator.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`bera-reth` and `bera-geth` exhibit inconsistent validation behaviour for Proof-of-Liquidity (PoL) transactions before the Prague1 fork is active. An attacker could exploit this discrepancy to create blocks that are accepted by one client but rejected by the other, potentially causing a consensus split.

In `bera-reth`'s implementation, PoL transaction validation is conditional on the Prague1 fork being active. Meaning that PoL transactions are allowed in blocks before Prague1:

bera-reth/src/consensus/mod.rs::validate_block_pre_execution()

```
fn validate_block_pre_execution(
    &self,
    block: &SealedBlock,
) -> Result<(), Self::Error> {
     as Consensus>::validate_block_pre_execution(
        &self.inner,
        block,
    )?;

    if self.chain_spec.is_prague1_active_at_timestamp(block.header().timestamp) {
        self.validate_pol_transaction(block)?;
    }
    Ok(())
}
```

In contrast, `bera-geth`'s block validator unconditionally rejects PoL transactions before Prague1:

bera-geth/core/block_validator.go::ValidateBody()

```
func (v *BlockValidator) ValidateBody(block *types.Block) error {
    // ...
    for i, tx := range block.Transactions() {
        // Berachain: validate the PoL tx is only the first tx in the block.
        switch {
        case isPrague1 && i == 0:
            if tx.Hash() != expectedPoLHash {
                return fmt.Errorf("PoL tx hash mismatch: have %v, want %v", tx.Hash(), expectedPoLHash)
            }
        case tx.Type() == types.PoLTxType:
            return fmt.Errorf("invalid block: tx at index %d is a PoL tx", i)
        }
```

This inconsistency means that before Prague1 activation, `bera-reth` would accept blocks containing PoL transactions (performing no validation on them), while `bera-geth` would reject such blocks entirely. An attacker could construct a block containing a PoL transaction before Prague1 is active, causing `bera-reth` nodes to accept the block whilst `bera-geth` nodes reject it, causing a chain split.

Recommendations

Modify `bera-reth` such that it rejects PoL transactions before Prague1.

Resolution

The development team has modified `bera-reth` to reject PoL transactions before Prague1 as per the recommendation.

This issue has been resolved in bera-reth#93.

BRG4-02	PoL Gas Limit Consensus Mismatch Between Clients		
Asset	`bera-reth/src/transaction/pol.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A consensus-breaking bug exists between the `bera-reth` and `bera-geth` execution clients due to conflicting gas limit values for Proof of Liquidity (PoL) transactions.

The `bera-reth` client uses the `EIP-7825` transaction gas limit cap of 16,777,216 gas (2^{24}), whilst the `bera-geth` client correctly implements the `BRIP-0004` specification requirement of 30,000,000 gas. This mismatch will cause the clients to generate PoL transactions with different gas limits, resulting in different transaction hashes and rendering blocks invalid across the network.

The issue occurs in the `create_pol_transaction()` function in `bera-reth`:

bera-reth/src/transaction/pol.rs::create_pol_transaction()

```
let pol_tx = PoLTx {
    chain_id: chain_spec.chain_id(),
    from: SYSTEM_ADDRESS,
    to: chain_spec.pol_contract(),
    input: Bytes::from(calldata),
    nonce,
    gas_limit: eip7825::TX_GAS_LIMIT_CAP, // @audit: Uses 16,777,216 instead of 30,000,000
    gas_price: base_fee.into(),
};
```

In contrast, `bera-geth` correctly uses the `BRIP-0004` compliant value:

bera-geth/params/protocol_params.go

```
PoLTxGasLimit uint64 = 30_000_000
```

The `BRIP-0004` specification explicitly defines the gas limit requirement in the PoLTx struct:

BRIP-0004 Specification

```
type PoLTx struct {
    ChainID  *big.Int
    From     common.Address  // Always SYSTEM_ADDRESS (0xfffffffffffffffffffffffffffffffffffffffe)
    To       common.Address  // PoL Distributor contract address
    Nonce    uint64          // MUST equal block_number - 1
    GasLimit uint64          // Set to 30,000,000 (system transaction gas limit)
    GasPrice *big.Int        // Set to base_fee for RPC compatibility
    Data     []byte          // distributeFor(bytes calldata pubkey) call data
}
```

Since PoL transactions are system-generated and must appear as the first transaction in every post-Prague1 block, this consensus bug will manifest immediately after the Prague1 fork activation. Any block containing a PoL transaction will be considered invalid by the opposing client, leading to a chain split.

The likelihood is high because PoL transactions are mandatory in every block after the Prague1 fork, and the impact is high due to the fundamental consensus failure that prevents network operation.

Recommendations

Change the gas limit value in `bera-reth` to match the `BRIP-0004` specification by replacing `eip7825::TX_GAS_LIMIT_CAP` with a constant value of 30,000,000.

Keep in mind that the transaction gas limit should not be changed to any other value as system calls in `revm` use a hard-coded value of 30 million gas.

Resolution

The development team has modified `bera-reth`'s PoL transaction gas limit to 30,000,000 as per the recommendation.

This issue has been resolved in bera-reth#53.

BRG4-03	`bera-reth` Missing RLP Length Validation During PoL Transaction Decoding		
Asset	`bera-reth/src/transaction/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `rlp_decode()` function for a PoL transaction does not validate that the actual buffer length matches the RLP header's declared payload length. As a result, `bera-reth` accepts malformed RLP data that `bera-geth` rejects, potentially leading to chain splits.

bera-reth/src/transaction/mod.rs::rlp_decode()

```
fn rlp_decode(buf: &mut &[u8]) -> alloy_rlp::Result<Self> {
    let header = alloy_rlp::Header::decode(buf)?;
    if !header.list {
        return Err(alloy_rlp::Error::UnexpectedString);
    }

    // @audit Missing length validation
    Ok(Self {
        chain_id: ChainId::decode(buf)?,
        from: Address::decode(buf)?,
        to: Address::decode(buf)?,
        nonce: u64::decode(buf)?,
        gas_limit: u64::decode(buf)?,
        gas_price: u128::decode(buf)?,
        input: Bytes::decode(buf)?,
    })
}
```

For example, any data with a longer payload than the header indicates would be accepted by `bera-reth` but is correctly rejected by `bera-geth`. An attacker can exploit this discrepancy to cause chain splits.

Recommendations

Implement proper RLP length validation in the `rlp_decode()` function. For example:

bera-reth/src/transaction/mod.rs::rlp_decode()

```rust
fn rlp_decode(buf: &mut &[u8]) -> alloy_rlp::Result<Self> {
  let header = alloy_rlp::Header::decode(buf)?;
  if !header.list {
    return Err(alloy_rlp::Error::UnexpectedString);
  }

  let remaining = buf.len();

  // Ensure payload is not shorter than indicated length
  if header.payload_length > remaining {
    return Err(alloy_rlp::Error::InputTooShort);
  }

  let decoded = Self {
    chain_id: ChainId::decode(buf)?,
    from: Address::decode(buf)?,
    to: Address::decode(buf)?,
    nonce: u64::decode(buf)?,
    gas_limit: u64::decode(buf)?,
    gas_price: u128::decode(buf)?,
    input: Bytes::decode(buf)?,
  };

  // Ensure indicated length matches decoded length
  if buf.len() + header.payload_length != remaining {
    return Err(alloy_rlp::Error::UnexpectedLength);
  }

  Ok(decoded)
}
```

Resolution

The development team has modified `bera-reth`'s PoL transaction decoding to validate the payload against the header's declared payload length as per the recommendation.

This issue has been resolved in bera-reth#120.

BRG4-04	Prague1 Empty Block Consensus Mismatch Between Clients		
Asset	`bera-geth/core/block_validator.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A critical consensus mismatch exists between `bera-geth` and `bera-reth` regarding empty block validation after the Prague1 upgrade. This discrepancy creates a potential for chain splits where the two clients would disagree on block validity.

After the Prague1 fork, every block must contain at least one transaction - specifically a Proof-of-Liquidity (PoL) transaction as the first transaction. However, the two implementations handle this requirement differently:

`bera-reth` correctly enforces the empty block check:

bera-reth/src/consensus/mod.rs::validate_pol_transaction()

```
if transactions.is_empty() {
    return Err(ConsensusError::Other(
        "Prague1 block must contain at least one PoL transaction".into(),
    ));
}
```

`bera-geth` performs transaction validation within a loop but lacks an explicit check for empty blocks:

bera-geth/core/block_validator.go::ValidateBody()

```
for i, tx := range block.Transactions() {
    // Berachain: validate the PoL tx is only the first tx in the block.
    switch {
    case isPrague1 && i == 0:
        if tx.Hash() != expectedPoLHash {
            return fmt.Errorf("PoL tx hash mismatch: have %v, want %v", tx.Hash(), expectedPoLHash)
        }
    case tx.Type() == types.PoLTxType:
        return fmt.Errorf("invalid block: tx at index %d is a PoL tx", i)
    }
    // ... more validation logic
}
```

If an empty block is proposed after Prague1, `bera-geth` will not enter the validation loop at all, effectively accepting the invalid empty block.

This creates a scenario where:

- `bera-reth` would reject an empty block with consensus error
- `bera-geth` would accept the same empty block as valid
- The chain would split, with each client following different forks

The `BRIP-0004` specification clearly shows an explicit check for empty blocks:

BRIP-0004 Specification

```
// Validate first transaction is PoL transaction
if len(block.Transactions()) == 0 || block.Transactions()[0].Type() != POL_TX_TYPE {
    return errors.New("post-prague1 block missing required PoL transaction")
}
```

Recommendations

Add an explicit empty block validation check in `validateBody()` before the transaction iteration loop.

bera-geth/core/block_validator.go::ValidateBody()

```
if isPrague1 {
    // Prague1 blocks must contain at least one PoL transaction
    if len(block.Transactions()) == 0 {
        return errors.New("Prague1 block must contain at least one PoL transaction")
    }

    polTx, err := types.NewPoLTx(
        v.config.ChainID,
        v.config.Berachain.Prague1.PoLDistributorAddress,
        new(big.Int).Sub(block.Number(), big.NewInt(1)),
        params.PoLTxGasLimit,
        block.BaseFee(),
        block.ProposerPubkey(),
    )
    // ... rest of existing logic
}
```

Resolution

The development team has modified `bera-geth`'s empty block validation to check for an empty block as per the recommendation.

This issue has been resolved in bera-geth#40.

BRG4-05	`bera-reth` PoL Transaction Sender Vector Misalignment		
Asset	`src/node/evm/assembler.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The PoL (Proof-of-Liquidity) transaction injection mechanism creates a critical misalignment between the `transactions` vector and the corresponding `senders` vector, corrupting transaction-sender mappings for all transactions after Prague1. This affects several RPC methods relying on the `senders` vector.

In the standard Reth block building flow, the `BasicBlockBuilder::finish()` function extracts both transactions and their corresponding senders from `Recovered<Transaction>` objects.

crates/evm/evm/src/execute.rs::finish()

```
let (transactions, senders) =
    self.transactions.into_iter().map(|tx| tx.into_parts()).unzip();

let block = self.assembler.assemble_block(BlockAssemblerInput {
    evm_env,
    execution_ctx: self.ctx,
    parent: self.parent,
    transactions,
    output: &result,
    bundle_state: &db.bundle_state,
    state_provider: &state,
    state_root,
})?;

let block = RecoveredBlock::new_unhashed(block, senders);
```

This creates parallel vectors where `transactions[i]` corresponds to `senders[i]`. However, Berachain's implementation breaks this invariant by injecting the PoL transaction during the `assemble_block()` phase rather than before sender extraction.

src/node/evm/assembler.rs::assemble_block()

```
if self.chain_spec.is_prague1_active_at_timestamp(timestamp) {
    let prev_proposer_pubkey = ctx.prev_proposer_pubkey.unwrap();

    // Synthesise PoL transaction and prepend to transactions list
    let pol_transaction = create_pol_transaction(
        self.chain_spec.clone(),
        prev_proposer_pubkey,
        evm_env.block_env.number,
        base_fee,
    )?;

    transactions.insert(0, pol_transaction); //@audit This breaks sender alignment
}
```

This results in:

1. The `finish()` function extracts `n` transactions and `n` senders from user transactions only

2. The `assemble_block()` function inserts the PoL transaction at index 0, creating `n+1` transactions but only `n` senders
3. When `RecoveredBlock::new_unhashed(block, senders)` is called, the misalignment causes:
    - `transactions[0]` = PoL transaction, but `senders[0]` = sender of first user transaction
    - `transactions[1]` = first user transaction, but `senders[1]` = sender of second user transaction
    - And so forth for all subsequent transactions

The impact and likelihood is medium because `BerachainPayloadBuilder` discards the `senders` vector, so its values are not used in the block building flow in `engine_forkchoiceUpdatedV3P11`. Hence, this issue will not result in a consensus bug or errors during block execution. The impact is limited to RPC methods that use the `senders` vector in their logic, such as `eth_simulateV1` and `eth_getBlockReceipts`.

Recommendations

Modify `finish()` to include the PoL transaction's caller address in the `senders` vector before `RecoveredBlock::new_unhashed()` is called.

Resolution

The development team has acknowledged this issue with the following comment:

> *"This issue only affects `eth_simulateV1` and `eth_getBlockReceipts` with the `pending` tag. This is currently considered an acceptable risk as these RPC methods are not consensus-critical. We plan to address this in a future update."*

BRG4-06	Engine API Fork Validation Discrepancy Between Clients		
Asset	`bera-reth/src/engine/rpc.rs, bera-geth/eth/catalyst/api.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

A discrepancy exists between `bera-reth` and `bera-geth` implementations of the Engine API forkchoice update methods. This inconsistency allows clients to accept or reject the same Engine API calls differently based on Prague1 fork activation, leading to an inconsistent user experience.

In `bera-reth`, both `fork_choice_updated_v3()` and `fork_choice_updated_v3_p11()` methods incorrectly call the same underlying implementation without performing fork-specific validation:

bera-reth/src/engine/rpc.rs::fork_choice_updated_v3()

```
async fn fork_choice_updated_v3(
    &self,
    fork_choice_state: ForkchoiceState,
    payload_attributes: Option,
) -> RpcResult {
    trace!(target: "rpc::engine", "Serving engine_forkchoiceUpdatedV3");
    Ok(self.inner.fork_choice_updated_v3_metered(fork_choice_state, payload_attributes).await?)
}

async fn fork_choice_updated_v3_p11(
    &self,
    fork_choice_state: ForkchoiceState,
    payload_attributes: Option,
) -> RpcResult {
    trace!(target: "rpc::engine", "Serving engine_forkchoiceUpdatedV3P11");
    Ok(self.inner.fork_choice_updated_v3_metered(fork_choice_state, payload_attributes).await?)
}
```

Both methods execute identical logic by calling `fork_choice_updated_v3_metered()`, when they should validate fork activation and proposer public key requirements differently.

In contrast, `bera-geth` correctly implements fork-specific validation:

bera-geth/eth/catalyst/api.go::ForkchoiceUpdatedV3()

```
func (api *ConsensusAPI) ForkchoiceUpdatedV3(update engine.ForkchoiceStateV1, params *engine.PayloadAttributes)
      ↪ (engine.ForkChoiceResponse, error) {
    if params != nil {
        switch {
        case params.ProposerPubkey != nil:
            return engine.STATUS_INVALID, attributesErr("unexpected proposer pubkey")
        case !api.checkFork(params.Timestamp, forks.Cancun, forks.Prague, forks.Osaka):
            return engine.STATUS_INVALID, unsupportedForkErr("fcuV3 must only be called for cancun or prague payloads")
        }
    }
    return api.forkchoiceUpdated(update, params, engine.PayloadV3, false)
}

func (api *ConsensusAPI) ForkchoiceUpdatedV3P11(update engine.ForkchoiceStateV1, params *engine.PayloadAttributes)
      ↪ (engine.ForkChoiceResponse, error) {
    if params != nil {
        switch {
        case params.ProposerPubkey == nil:
            return engine.STATUS_INVALID, attributesErr("missing proposer pubkey")
        case !api.checkFork(params.Timestamp, forks.Prague1):
            return engine.STATUS_INVALID, unsupportedForkErr("fcuV3P11 must only be called for prague1 payloads")
        }
    }
    return api.forkchoiceUpdated(update, params, engine.PayloadV3P11, false)
}
```

The discrepancy manifests in several ways:

- `bera-geth` rejects `forkchoiceUpdatedV3` calls after Prague1 activation
- `bera-geth` accepts `forkchoiceUpdatedV3P11` calls only after Prague1 activation
- `bera-reth` accepts both calls regardless of fork activation
- Different proposer public key validation between the two clients

While `bera-reth` does perform validation deeper in the call stack during block execution via `validate_proposer_pubkey_prague1()`, this late validation creates an inconsistent user experience.

This represents a low-impact issue due to the validation being performed later in the call stack, with medium likelihood as it affects routine Engine API operations around the Prague1 fork.

Recommendations

Implement proper fork validation in `bera-reth`'s Engine API methods to match `bera-geth`'s behaviour:

1. Add fork activation validation to `fork_choice_updated_v3()` to reject calls when Prague1 is active
2. Add fork activation validation to `fork_choice_updated_v3_p11()` to only accept calls when Prague1 is active
3. Implement proposer public key validation at the API level rather than deferring to block execution

Resolution

The development team has added fork activation and proposer public key validation to `engine_forkchoiceUpdatedV3` and `engine_forkchoiceUpdatedV3P11`.

This issue has been resolved in bera-reth#63 and bera-reth#122.

BRG4-07	`bera-geth` PoL Transaction Simulation Bypasses System Address Check		
Asset	`bera-geth/core/types/tx_pol.go, bera-geth/internal/ethapi/transaction_args.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `IsPoLDistribution()` function in `tx_pol.go` determines whether a transaction is a Proof-of-Liquidity (PoL) distribution transaction by checking only the recipient address and transaction data, but fails to verify that the sender is the system address (`0xfffffffffffffffffffffffffffffffffffffffe`).

bera-geth/core/types/tx_pol.go::IsPoLDistribution()

```
// IsPoLDistribution returns true if the transaction is a PoL distribution.
func IsPoLDistribution(to *common.Address, data []byte, distributorAddress common.Address) bool {
    // Txs that call the `distributeFor(bytes pubkey)` method on the PoL Distributor
    // contract are also consideredPoL txs.
    return to != nil && *to == distributorAddress && isDistributeForCall(data)
}
```

This function is used in the RPC layer in `transaction_args.go` to determine if a transaction should be treated as a PoL transaction during simulation operations. When `IsPoLDistribution()` returns `true`, the transaction is marked with `IsPoLTx: true` in the `Message` struct:

bera-geth/internal/ethapi/transaction_args.go::ToMessage()

```
func (args *TransactionArgs) ToMessage(...) *core.Message {
    // ... snip ...
    return &core.Message{
        // ... snip ...
        IsPoLTx: isPrague1 && types.IsPoLDistribution(args.To, args.data(), distributorAddress),
    }
}
```

This triggers special handling via `ApplyPoLMessage()` that bypasses normal transaction validation including signature verification.

Any user can craft a transaction calling the `distributeFor()` method on the `Distributor.sol` contract and have it incorrectly classified as a PoL transaction during simulation. This affects the following JSON-RPC methods:

1. `eth_call`:

   bera-geth/internal/ethapi/api.go::doCall()

   ```
   // @audit The message will be treated as a PoL transaction and result in incorrect simulation
   msg := args.ToMessage(header.BaseFee, skipChecks, skipChecks,
       b.ChainConfig().IsPrague1(header.Number, header.Time),
       b.ChainConfig().Berachain.Prague1.PoLDistributorAddress)
   ```

2. `eth_estimateGas`:

   bera-geth/internal/ethapi/api.go::DoEstimateGas()

   ```
   // @audit The message will be treated as a PoL transaction and result in incorrect simulation
   call := args.ToMessage(header.BaseFee, true, true,
       b.ChainConfig().IsPrague1(header.Number, header.Time),
       b.ChainConfig().Berachain.Prague1.PoLDistributorAddress)
   ```

3. `eth_createAccessList`:

bera-geth/internal/ethapi/api.go::CreateAccessList()

```
// @audit The message will be treated as a PoL transaction and result in a potentially incorrect access list
msg := args.ToMessage(header.BaseFee, true, true, isPrague1, distributorAddress)
```

4. `eth_sendTransaction` and `eth_signTransaction`:

bera-geth/internal/ethapi/transaction_args.go::ToTransaction()

```
// @audit This will treat the transaction as a PoL transaction even if the sender is not the system address
case isPrague1 && types.IsPoLDistribution(args.To, args.data(), distributorAddress):
    usedType = types.PoLTxType
```

While legitimate PoL transactions in blocks are properly validated using the transaction type (`tx.Type() == types.PoLTxType`) rather than this function, the incorrect simulation behaviour could mislead users and applications relying on these RPC methods for gas estimation and transaction simulation.

Recommendations

Add a sender address parameter to the `IsPoLDistribution()` function and verify that the sender is the system address (`params.SystemAddress`) before classifying a transaction as a PoL transaction.

Update all call sites to pass the sender address.

Resolution

The development team has modified `bera-geth`'s `IsPoLDistribution()` function to verify the sender address as per the recommendation.

This issue has been resolved in bera-geth#52.

BRG4-08	`bera-geth ethclient` PoL Transaction Signature Validation Failure		
Asset	`bera-geth/ethclient/ethclient.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `ethclient` methods `TransactionByHash()` and `TransactionInBlock()` incorrectly reject PoL transactions due to signature validation checks that are incompatible with the signature-less nature of PoL transactions.

PoL transactions are system-generated transactions that originate from the system address and carry no signature by design. The `PoLTx` type implements `rawSignatureValues()` as a no-op function that returns `nil` values:

bera-geth/core/types/tx_pol.go::rawSignatureValues()

```
// No-op: PoLTx is originated from the system address and carries no signature.
func (*PoLTx) rawSignatureValues() (v, r, s *big.Int) {
    return nil, nil, nil
}
```

However, both `TransactionByHash()` and `TransactionInBlock()` methods in the `ethclient` package perform signature validation that fails for PoL transactions:

bera-geth/ethclient/ethclient.go::TransactionByHash()

```
} else if _, r, _ := json.tx.RawSignatureValues(); r == nil {
    return nil, false, errors.New("server returned transaction without signature")
}
```

bera-geth/ethclient/ethclient.go::TransactionInBlock()

```
} else if _, r, _ := json.tx.RawSignatureValues(); r == nil {
    return nil, errors.New("server returned transaction without signature")
}
```

This causes these methods to fail when attempting to retrieve PoL transactions, returning an error instead of the expected transaction data. The validation assumes all transactions must have signatures, which contradicts the PoL transaction design.

These methods are part of the public `TransactionReader` and `ChainReader` interfaces that external applications might use to interact with the blockchain. External applications using these interfaces to query transaction data would be unable to retrieve PoL transactions.

This issue is classified as low impact as these methods are not used within the `bera-geth` codebase itself. It would only affect external applications that specifically attempt to retrieve PoL transactions using these client methods.

Recommendations

Modify the signature validation logic in `TransactionByHash()` and `TransactionInBlock()` to account for PoL transactions. The validation should either:

1. Skip signature validation for PoL transaction types

2. Check the transaction type before applying signature validation rules
3. Handle signature-less transactions as a valid case for system transactions

Consider implementing a transaction type-aware validation approach:

```
if _, r, _ := json.tx.RawSignatureValues(); r == nil && json.tx.Type() != types.PoLTxType {
    return nil, false, errors.New("server returned transaction without signature")
}
```

Resolution

The development team has modified the `rawSignatureValues()` method for the `PoLTx` type to return zero values instead of `nil`. This is now compatible with the validation logic in `TransactionByHash()` and `TransactionInBlock()`.

This issue has been resolved in bera-geth#37.

BRG4-09	`bera-reth` PoL Transaction Trait Implementation Is Incorrect		
Asset	`bera-reth/src/transaction/mod.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `PoLTx` implementation of the alloy `Transaction` trait contains two incorrect method implementations that deviate from the established specifications for legacy transactions. According to the `alloy` specification and reference implementations in `revm`, legacy transactions should behave differently than dynamic fee transactions.

The two incorrect implementations are:

1. `max_priority_fee_per_gas()` incorrectly returns `Some(self.gas_price)`:

   bera-reth/src/transaction/mod.rs::PoLTx::max_priority_fee_per_gas()

   ```
   fn max_priority_fee_per_gas(&self) -> Option {
       Some(self.gas_price)
   }
   ```

   According to the `alloy` trait documentation: "For dynamic fee transactions returns the Priority fee the caller is paying to the block author. This will return `None` for legacy fee transactions." Since PoL transactions are legacy transactions (`is_dynamic_fee()` returns `false`), this should return `None` .

2. `effective_gas_price()` incorrectly returns `0`:

   bera-reth/src/transaction/mod.rs::PoLTx::effective_gas_price()

   ```
   fn effective_gas_price(&self, _base_fee: Option) -> u128 {
       0
   }
   ```

   The `alloy` trait specification states: "Returns effective gas price is gas price field for Legacy and Eip2930 transaction." For legacy transactions, this should return `self.gas_price` and ignore the `base_fee` parameter entirely.

These implementations contradict the behaviour found in `bera-geth`:

bera-geth/core/types/tx_pol.go::effectiveGasPrice()

```
func (tx *PoLTx) effectiveGasPrice(dst *big.Int, baseFee *big.Int) *big.Int {
    return dst.Set(tx.GasPrice)
}
```

These implementation differences do not affect consensus or execution behaviour, and hence are classified aslow impact and likelihood. PoL transactions use their own transaction type (`0x7E`) in `revm` execution, which bypasses the legacy transaction logic where these trait methods would matter. The `effective_gas_price()` returning 0 is actually correct per `BRIP-0004` specification since PoL transactions don't consume gas, though it's not what the `alloy` trait specification expects. The main impact is API inconsistency when tools expect standard `alloy` `Transaction` trait behaviour for legacy transactions.

Recommendations

Consider correcting the `PoLTx` trait implementation to match the alloy specification for legacy transactions for API consistency:

1. Change `max_priority_fee_per_gas()` to return `None` to match legacy transaction behaviour:

   bera-reth/src/transaction/mod.rs::PoLTx::max_priority_fee_per_gas()

   ```
   fn max_priority_fee_per_gas(&self) -> Option {
       None
   }
   ```

2. Change `effective_gas_price()` to return `self.gas_price` to match legacy transaction behaviour:

   bera-reth/src/transaction/mod.rs::PoLTx::effective_gas_price()

   ```
   fn effective_gas_price(&self, _base_fee: Option) -> u128 {
       self.gas_price
   }
   ```

However, the `effective_gas_price()` implementation returning 0 appears to be intentionally correct per `BRIP-0004` specification since PoL transactions don't consume gas. If this is intended behaviour, correct `bera-geth`'s `effectiveGasPrice()` to return `0` instead of `tx.GasPrice`.

Resolution

The development team has modified `bera-reth`'s `PoLTx` implementation to be consistent with the `alloy` specification for dynamic fee transactions instead of legacy transactions.

This issue has been resolved in bera-reth#55 and bera-reth#151.

BRG4-10	`bera-geth` PoL Transaction Error Mutation Breaks Error Type Checking		
Asset	`bera-geth/core/state_transition.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `ApplyPoLMessage()` function in `core/state_transition.go` wraps standard EVM errors with `fmt.Errorf()`, breaking error type checking throughout the codebase:

bera-geth/core/state_transition.go::ApplyPoLMessage()

```
func ApplyPoLMessage(msg *Message, evm *vm.EVM) *ExecutionResult {
    evm.StateDB.AddAddressToAccessList(*msg.To)

    result := &ExecutionResult{}
    ret, leftOverGas, err := evm.Call(msg.From, *msg.To, msg.Data, msg.GasLimit, common.U256(0))
    if err != nil {
        // @audit EVM errors are wrapped with fmt.Errorf()
        result.Err = fmt.Errorf("PoL tx failed to execute: %v", err)
    }
    result.ReturnData = ret
    result.MaxUsedGas = msg.GasLimit - leftOverGas
    return result
}
```

This breaks Go's error type checking mechanism (`errors.Is()`), which is used in multiple locations to handle specific error types:

1. Gas Estimation (`eth/gasestimator/gasestimator.go`)

   bera-geth/eth/gasestimator/gasestimator.go

   ```
   if result != nil && !errors.Is(result.Err, vm.ErrOutOfGas) {
       return 0, result.Revert(), result.Err
   }
   ```

   The check will not properly detect out-of-gas errors for failed PoL transactions, potentially affecting gas estimation logic.

2. API Simulation (`internal/ethapi/simulate.go`)

   bera-geth/internal/ethapi/simulate.go

   ```
   if errors.Is(result.Err, vm.ErrExecutionReverted) {
       // If the result contains a revert reason, try to unpack it.
       revertErr := newRevertError(result.Revert())
       callRes.Error = &callError{Message: revertErr.Error(), Code: errCodeReverted, Data: revertErr.ErrorData().(string)}
   } else {
       callRes.Error = &callError{Message: result.Err.Error(), Code: errCodeVMError}
   }
   ```

   Reverted PoL transactions will be classified as `errCodeVMError` instead of `errCodeReverted`, losing revert reason data.

3. API Calls (`internal/ethapi/api.go`)

bera-geth/internal/ethapi/api.go

```
if errors.Is(result.Err, vm.ErrExecutionReverted) {
    return nil, newRevertError(result.Revert())
}
return result.Return(), result.Err
```

Reverted PoL transactions will not return structured revert errors, affecting client error handling.

The wrapped error will not match these type checks, causing these functions to handle PoL transaction failures incorrectly. All these locations expect standard EVM errors but receive wrapped errors that break type checking when PoL transactions fail.

However, the impact is limited since:

- PoL transactions are system-generated, not user-initiated
- These errors primarily affect logging and error reporting rather than consensus
- The transaction failure itself is still properly handled (the block continues processing)

Recommendations

Replace the error wrapping with a structured approach that preserves the original error type.

Alternatively, if additional context is needed, use Go's error wrapping with `%w` to maintain the error chain for type checking.

Resolution

The development team has modified the `ApplyPoLMessage()` function to preserve the error chain using `%w`.

This issue has been resolved in `bera-geth` at commit ac06e70.

BRG4-11	`bera-geth` Incorrect PoL Transaction Gas Price
Asset	`bera-geth/core/types/tx_pol.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `gasTipCap()` method for PoL transactions incorrectly returns `tx.GasPrice` instead of zero, leading to incorrect gas price calculations during transaction to message conversion.

In `EIP-1559` transactions, the gas tip cap represents the priority fee (tip) that users are willing to pay on top of the base fee. For PoL transactions, since they are system-generated transactions that pay no gas fees and have their gas price set to the base fee, the priority fee should be zero.

The current implementation in `bera-geth/core/types/tx_pol.go` incorrectly returns the full gas price:

bera-geth/core/types/tx_pol.go::gasTipCap()

```
func (tx *PoLTx) gasTipCap() *big.Int { return tx.GasPrice }
```

Since `tx.GasPrice` is set to the base fee, this means `gasTipCap()` returns the base fee instead of zero.

This affects the gas price calculation in `TransactionToMessage()` at `bera-geth/core/state_transition.go`:

bera-geth/core/state_transition.go::TransactionToMessage()

```
msg.GasPrice = msg.GasPrice.Add(msg.GasTipCap, baseFee)
```

With the incorrect implementation, this becomes `baseFee + baseFee + baseFee = 3 * baseFee` whereas the intended gas price is just the `baseFee`.

However, the practical impact is minimal because PoL transactions do not actually pay gas fees, and the gas price gets corrected due to the `gasFeeCap` check below:

bera-geth/core/state_transition.go::TransactionToMessage()

```
if msg.GasPrice.Cmp(msg.GasFeeCap) > 0 {
    // @audit This will set the gas price back to the base fee
    msg.GasPrice = msg.GasFeeCap
}
```

Since `gasFeeCap()` also returns `tx.GasPrice` (the base fee), the final gas price is capped back to the base fee, effectively correcting the error.

Recommendations

Update the `gasTipCap()` method in the `PoLTx` type to return zero instead of the gas price:

bera-geth/core/types/tx_pol.go::gasTipCap()

```
func (tx *PoLTx) gasTipCap() *big.Int { return new(big.Int) }
```

This change would ensure that PoL transactions correctly represent a zero priority fee, making the gas price calculation semantically correct and preventing potential issues if the gas fee cap correction logic is modified in the future.

Resolution

The development team has modified the `gasTipCap()` method to return zero instead of the gas price as recommended.

This issue has been resolved in `bera-geth` at commit 7f6ad26.

BRG4-12	`bera-reth` Payload ID Is Missing `prev_proposer_pubkey`
Asset	`bera-reth/src/engine/payload.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `payload_id()` function in `bera-reth` does not include the Berachain-specific `prev_proposer_pubkey` field when generating payload identifiers, creating a potential for payload ID collisions with pre-Prague1 payloads.

In `try_new()`, the code calls the standard `reth` `payload_id()` function which only hashes the parent block hash, timestamp, `prev_randao`, `suggested_fee_recipient`, `withdrawals`, and `parent_beacon_block_root`. However, Berachain adds a critical `prev_proposer_pubkey` field that is stored in the payload attributes but not included in the ID calculation.

bera-reth/src/engine/payload.rs::BerachainPayloadBuilderAttributes::try_new()

```
fn try_new(
    parent: B256,
    attributes: Self::RpcPayloadAttributes,
    _version: u8,
) -> Result
where
    Self: Sized,
{
    // @audit The payload id doesn't include the prev_proposer_pubkey
    let payload_id = payload_id(&parent, &attributes.inner);
    Ok(Self {
        id: payload_id,
        parent,
        // ... snip ...
        prev_proposer_pubkey: attributes.prev_proposer_pubkey,
    })
}
```

This creates a scenario where two payloads with identical Ethereum parameters but different `prev_proposer_pubkey` values will generate the same `PayloadId`.

However, this theoretical collision is extremely unlikely to occur in practice as it requires two almost-identical payloads to be built at the same time. Hence, this issue is considered as informational severity.

Recommendations

Implement a custom `payload_id()` function for Berachain that includes the `prev_proposer_pubkey` field in the hash calculation. This ensures that payloads with different proposer public keys generate unique identifiers.

Then update `BerachainPayloadBuilderAttributes::try_new()` to use this custom function instead of the standard Ethereum `payload_id()`.

Resolution

The development team has implemented a new `berachain_payload_id()` function that includes the `prev_proposer_pubkey` field in the hash calculation.

This issue has been resolved in bera-reth#72.

BRG4-13	`bera-geth` PoL Transaction JSON Marshalling Missing Case
Asset	`bera-geth/core/types/transaction_marshalling.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

PoL transactions cannot be marshalled to JSON due to a missing case in the `MarshalJSON()` method. While the `UnmarshalJSON()` method at `transaction_marshalling.go` includes a `PoLTxType` case for deserialisation, the corresponding `MarshalJSON()` method lacks any handling for PoL transactions.

The `MarshalJSON()` method contains cases for all other transaction types (`LegacyTx` , `AccessListTx` , `DynamicFeeTx` , `BlobTx` , `SetCodeTx`) but completely omits `*PoLTx` . When a PoL transaction is marshalled to JSON, only the basic `Hash` and `Type` fields are populated:

bera-geth/core/types/transaction_marshalling.go::MarshalJSON()

```
// These are set for all tx types.
enc.Hash = tx.Hash()
enc.Type = hexutil.Uint64(tx.Type())
```

All other fields such as `ChainID` , `Nonce` , `To` , `Gas` , `GasPrice` , and `Input` remain unpopulated in the JSON output, creating an incomplete and potentially unusable JSON representation.

This asymmetry between marshalling and unmarshalling capabilities creates functional inconsistencies. Applications that need to serialise PoL transactions to JSON for logging, debugging, API responses, or persistence will receive incomplete data structures that cannot be properly round-tripped through JSON serialisation.

Recommendations

Add a `*PoLTx` case to the `MarshalJSON()` method to properly serialise PoL transaction fields. The implementation should populate the relevant fields such as `ChainID` , `Nonce` , `To` , `Gas` , `GasPrice` , `Value` (zero), and `Input` to ensure complete JSON representation and proper round-trip serialisation capabilities.

Resolution

The development team has added a `*PoLTx` case to the `MarshalJSON()` method to properly serialise PoL transaction fields.

This issue has been resolved in bera-geth#39 and bera-geth#84.

BRG4-14	`bera-geth` PoL Transaction JSON Marshalling Redundant Value Check
Asset	`bera-geth/core/types/transaction_marshalling.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

PoL transactions require a `value` field during JSON unmarshalling despite having no conceptual value and always returning zero. The `UnmarshalJSON()` method at `transaction_marshalling.go` enforces this requirement:

bera-geth/core/types/transaction_marshalling.go::UnmarshalJSON()

```
case PolTxType:
    // .. snip ..
    if dec.Value == nil {
        return errors.New("missing required field 'value' in transaction")
    }
    // .. snip ..
```

However, PoL transactions are designed to have no value transfer capabilities. The `PoLTx.value()` method in `tx_pol.go` explicitly returns `new(big.Int)` (zero), confirming that PoL transactions have a hardcoded zero value:

bera-geth/core/types/tx_pol.go::value()

```
func (*PoLTx) value() *big.Int { return new(big.Int) }
```

This creates an inconsistency where the JSON representation requires a field that is conceptually meaningless for PoL transactions. The validation check serves no functional purpose since the value will always be ignored and set to zero regardless of what is provided in the JSON.

Recommendations

Remove the value field requirement check for PoL transactions in the `UnmarshalJSON()` method. The check is redundant since PoL transactions have a hardcoded zero value regardless of input. This will simplify the JSON interface and remove the misleading requirement for a meaningless field.

Resolution

The development team has removed the value field requirement check for PoL transactions in the `UnmarshalJSON()` method.

This issue has been resolved in `bera-geth` at commit 9d70186.

BRG4-15	`bera-geth` PoL Transaction Should Return Error For `nil` Pubkey
Asset	`bera-geth/core/types/tx_pol.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `getDistributeForData()` function in `bera-geth/core/types/tx_pol.go` accepts a `nil` pubkey parameter and silently creates transaction calldata with an empty pubkey instead of returning an error.

bera-geth/core/types/tx_pol.go::getDistributeForData()

```
func getDistributeForData(pubkey *common.Pubkey) ([]byte, error) {
    var pubkeyBytes []byte
    if pubkey == nil {
        // @audit This should return an error instead of using an empty pubkey
        pubkeyBytes = common.Pubkey{}.Bytes()
    } else {
        pubkeyBytes = pubkey.Bytes()
    }
    // ... rest of function
}
```

This function is used to create PoL transactions, which should only be generated in post-Prague1 blocks where a valid parent proposer pubkey is always available. The `ParentProposerPubkey` field in block headers is optional and will be `nil` in pre-Prague1 blocks.

Whilst the current code paths only call `NewPoLTx()` after checking `IsPrague1()`, accepting `nil` pubkeys without error creates a potential inconsistency. If a PoL transaction were somehow created with a `nil` pubkey, it would generate invalid calldata for the `distributeFor(bytes pubkey)` method with an empty pubkey parameter.

Recommendations

Implement defensive programming by returning an error when the pubkey is `nil`:

bera-geth/core/types/tx_pol.go::getDistributeForData()

```
func getDistributeForData(pubkey *common.Pubkey) ([]byte, error) {
    if pubkey == nil {
        return nil, errors.New("pubkey cannot be nil for PoL transaction")
    }

    pubkeyBytes := pubkey.Bytes()
    arguments, err := distributeForMethod.Inputs.Pack(pubkeyBytes)
    if err != nil {
        return nil, err
    }
    return append(distributeForMethod.ID, arguments...), nil
}
```

This ensures that any unexpected `nil` pubkey scenarios are caught early with a clear error message rather than silently generating invalid transaction data.

Resolution

The development team has implemented defensive programming by returning an error when the pubkey is `nil`.

This issue has been resolved in `bera-geth` at commit 1301bf6.

BRG4-16	`bera-geth` Engine API Fork Validation Bypass After Osaka Activation
Asset	`bera-geth/eth/catalyst/api.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `NewPayloadV4()` and `ForkchoiceUpdatedV3()` functions contain incorrect fork validation logic that allows these deprecated methods to be called after the Osaka fork activation, when they should be rejected in favour of their `P11` variants.

Both functions use the following validation pattern:

bera-geth/eth/catalyst/api.go::NewPayloadV4()

```
case !api.checkFork(params.Timestamp, forks.Prague, forks.Osaka):
    return invalidStatus, unsupportedForkErr("newPayloadV4 must only be called for prague payloads")
```

The `checkFork()` function determines the latest active fork using `LatestFork(timestamp)` and checks if it matches any of the provided forks:

bera-geth/eth/catalyst/api.go::checkFork()

```
func (api *ConsensusAPI) checkFork(timestamp uint64, forks ...forks.Fork) bool {
  latest := api.config().LatestFork(timestamp)
  for _, fork := range forks {
    if latest == fork {
      return true
    }
  }
  return false
}
```

The `LatestFork()` function returns forks in priority order: Osaka > Prague1 > Prague > Cancun:

bera-geth/params/config.go::LatestFork()

```
func (c *ChainConfig) LatestFork(time uint64) forks.Fork {
  // Assume last non-time-based fork has passed.
  london := c.LondonBlock

  switch {
  case c.IsOsaka(london, time):
    return forks.Osaka
  case c.IsPrague1(london, time):
    return forks.Prague1
  case c.IsPrague(london, time):
    return forks.Prague
  case c.IsCancun(london, time):
    return forks.Cancun
  // ... snip ...
  }
}
```

When Osaka is activated, `LatestFork()` returns `forks.Osaka`, which matches the second parameter in the `checkFork()` call. This causes the validation to pass when it should fail, allowing deprecated API methods to continue functioning post-Osaka.

This bug would cause an issue where the `P11` methods are not available after Osaka activation. Since Osaka is a future fork, there is currently no impact, but addressing this issue now will prevent future complications in the case that this issue still has not been addressed during the Osaka activation.

Recommendations

Remove `forks.Osaka` from the fork validation in both `NewPayloadV4()` and `ForkchoiceUpdatedV3()` functions, and add `forks.Osaka` to the corresponding `P11` functions:

For `NewPayloadV4()`:

bera-geth/eth/catalyst/api.go::NewPayloadV4()

```
case !api.checkFork(params.Timestamp, forks.Prague):
    return invalidStatus, unsupportedForkErr("newPayloadV4 must only be called for prague payloads")
```

For `ForkchoiceUpdatedV3()`:

bera-geth/eth/catalyst/api.go::ForkchoiceUpdatedV3()

```
case !api.checkFork(params.Timestamp, forks.Cancun, forks.Prague):
    return engine.STATUS_INVALID, unsupportedForkErr("fcuV3 must only be called for cancun or prague payloads")
```

Additionally, update the `P11` functions to support Osaka:

For `NewPayloadV4P11()`:

bera-geth/eth/catalyst/api.go::NewPayloadV4P11()

```
case !api.checkFork(params.Timestamp, forks.Prague1, forks.Osaka):
    return invalidStatus, unsupportedForkErr("newPayloadV4P11 must only be called for prague1 and osaka payloads")
```

For `ForkchoiceUpdatedV3P11()`:

bera-geth/eth/catalyst/api.go::ForkchoiceUpdatedV3P11()

```
case !api.checkFork(params.Timestamp, forks.Prague1, forks.Osaka):
    return engine.STATUS_INVALID, unsupportedForkErr("fcuV3P11 must only be called for prague1 and osaka payloads")
```

This ensures that after Osaka activation, the deprecated methods are correctly rejected and clients must use the `P11` variants which will continue to function across both Prague1 and Osaka forks.

Resolution

The development team has removed `forks.Osaka` from the fork validation in both `NewPayloadV4()` and `ForkchoiceUpdatedV3()` functions, and added `forks.Osaka` to the corresponding `P11` functions.

This issue has been resolved in bera-geth#37.

BRG4-17	Default Configuration Discrepancies Between Clients
Asset	`bera-geth/params/config.go, bera-reth/src/chainspec/mod.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

Multiple default configuration discrepancies exist between `bera-geth` and `bera-reth` that could lead to consensus issues if nodes operate without proper genesis file configuration. While these issues should not affect properly configured production networks that use official genesis files, they represent implementation inconsistencies that could cause problems during development or misconfiguration scenarios.

Prague1 Fork Timing Discrepancy

The default Prague1 fork activation time differs significantly between implementations:

- `bera-geth`: `999999999999999` (effectively never activated by default)
- `bera-reth`: `0` (activated at genesis by default)

This is evident in `bera-geth/params/config.go` where the Prague1 time is set to an extremely large number:

bera-geth/params/config.go::BerachainChainConfig

```
Prague1: Prague1Config{
    Time:                   newUint64(999999999999999),
    MinimumBaseFeeWei:      1000000000, // 1 gwei
    BaseFeeChangeDenominator: 48,        // 6x increase from the default
},
```

While `bera-reth` uses the genesis configuration directly from `berachain_genesis_config.prague1.time`.

Blob Configuration Mismatch

Berachain's Prague fork blob configuration incorrectly matches Cancun parameters instead of standard Prague parameters:

In `bera-geth/params/config.go`, `DefaultBerachainPragueBlobConfig` uses:

bera-geth/params/config.go::DefaultBerachainPragueBlobConfig

```
// @audit Target: 3, Max: 6, UpdateFraction: 3338477 (Cancun values)
DefaultBerachainPragueBlobConfig = &BlobConfig{
    Target:         3,
    Max:            6,
    UpdateFraction: 3338477,
}
```

While standard Prague configuration in `bera-geth/params/config.go` uses:

bera-geth/params/config.go::DefaultPragueBlobConfig

```
// @audit Target: 6, Max: 9, UpdateFraction: 5007716
DefaultPragueBlobConfig = &BlobConfig{
    Target:         6,
    Max:            9,
    UpdateFraction: 5007716,
}
```

This discrepancy would immediately cause consensus splits when processing blocks containing blob transactions, as the `CalcBlobFee()` function would produce different results between clients using different configurations, as `bera-reth` uses the Prague blob schedule by default.

Minimum Base Fee Specification Violation

Both implementations default to 1 gwei minimum base fee (`MinimumBaseFeeWei: 1000000000`), but `BRIP-0002` specifies 10 gwei as the required minimum. This creates a discrepancy with the official Berachain specification, though both clients are consistently wrong.

Recommendations

Align default configurations between both clients and with official specifications:

1. **Standardise Prague1 timing**: Use consistent default values or ensure both clients handle undefined values identically
2. **Fix blob configuration**: Update `DefaultBerachainPragueBlobConfig` in `bera-geth` to match Prague standards rather than Cancun:

   bera-geth/params/config.go

   ```
   DefaultBerachainPragueBlobConfig = &BlobConfig{
       Target:         6,
       Max:            9,
       UpdateFraction: 5007716,
   }
   ```

3. **Correct minimum base fee**: Update both clients to use 10 gwei as specified in `BRIP-0002` :

   bera-geth/params/config.go

   ```
   MinimumBaseFeeWei: 10000000000, // 10 gwei
   ```

Resolution

The development team has acknowledged this issue with the following comment:

> *"The blob configuration and minimum base fee values are intentional. Berachain uses the Cancun blob parameters for Prague and does not support the standard Prague blob parameters. BRIP-0002 has been updated to support a minimum base fee of 1 gwei instead of 10 gwei. Given that we provide a genesis file that overrides the default values, the discrepancies are not an issue."*

BRG4-18	`bera-reth` Prague1 Fork Validation Missing Prague Check
Asset	`bera-reth/src/chainspec/mod.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The Prague1 hardfork validation logic in `BerachainChainSpec::from()` does not verify that the Prague hardfork is configured when Prague1 is present. The current validation only checks temporal ordering when both forks are configured, but allows Prague1 to be configured without Prague being present.

In `bera-reth/src/chainspec/mod.rs`, the validation logic uses a match pattern that permits `prague_time` to be `None` while `prague1_time` exists:

bera-reth/src/chainspec/mod.rs::BerachainChainSpec::from()

```
match (genesis.config.prague_time, berachain_genesis_config.prague1.time) {
    (Some(prague_time), prague1_time) if prague1_time < prague_time => {
        panic!(
            "Prague1 hardfork must activate at or after Prague hardfork..."
        );
    }
    _ => {}  // This allows (None, prague1_time) case
}
```

This validation gap violates the logical dependency that Prague1 should only be configured if Prague is also configured, since Prague1 is conceptually a follow-up to the Prague hardfork. Allowing Prague1 without Prague creates an inconsistent fork configuration that could lead to unexpected network behaviour or consensus failures.

Recommendations

Add validation to ensure Prague is configured when Prague1 is present. Modify the match pattern to explicitly handle the `(None, _)` case:

bera-reth/src/chainspec/mod.rs::BerachainChainSpec::from()

```
match (genesis.config.prague_time, berachain_genesis_config.prague1.time) {
    (Some(prague_time), prague1_time) if prague1_time < prague_time => {
        panic!(
            "Prague1 hardfork must activate at or after Prague hardfork. Prague time: {prague_time}, Prague1 time: {prague1_time}.
                ↪  Check that Prague1 time is not malformed (should be a valid Unix timestamp)."
        );
    }
    (None, _) => {
        panic!("Prague1 hardfork requires Prague hardfork to be configured");
    }
    _ => {}
}
```

Resolution

The development team has modified the validation logic to explicitly handle the `(None, _)` case as recommended.

This issue has been resolved in bera-reth#121.

BRG4-19	`bera-reth` Unmaintained Dependencies
Asset	`code/bera-reth`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `bera-reth` project contains dependencies on unmaintained crates that pose long-term security and reliability risks.

The `derivative` and `paste` crates are flagged as unmaintained under advisory RUSTSEC-2024-0388, RUSTSEC-2024-0436 respectively. While not immediately exploitable, unmaintained dependencies will not receive security patches, bug fixes, or compatibility updates for newly discovered issues.

Full details can be found by running `cargo audit`.

Recommendations

Run `cargo audit` regularly to monitor unmaintained dependencies and security advisories, and consider replacing these crates by maintained alternatives.

Resolution

The development team has acknowledged this issue with the following comment:

> *"These crates come from `reth`. We cannot replace them."*

BRG4-20	Miscellaneous General Comments
Asset	All contracts
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. **Hardcoded Block Time Duration**
   Related Asset(s): bera-reth/src/node/evm/config.rs
   The `build_pending_env()` function contains a hardcoded magic number for Berachain's block time duration. The timestamp calculation uses `parent.timestamp().saturating_add(2)` where the value `2` represents the 2-second block time for Berachain networks.

   bera-reth/src/node/evm/config.rs::build_pending_env()

   ```
   impl BuildPendingEnv for BerachainNextBlockEnvAttributes {
       fn build_pending_env(parent: &SealedHeader) -> Self {
           Self {
               timestamp: parent.timestamp().saturating_add(2), //@audit hardcoded block time
               suggested_fee_recipient: parent.beneficiary(),
               prev_randao: B256::random(),
               gas_limit: parent.gas_limit(),
               parent_beacon_block_root: parent.parent_beacon_block_root().map(|_| B256::ZERO),
               withdrawals: parent.withdrawals_root().map(|_| Default::default()),
               prev_proposer_pubkey: parent.header().prev_proposer_pubkey,
           }
       }
   }
   ```

   This hardcoded value reduces code maintainability and clarity. If the block time needs to be modified in the future, developers must locate this magic number within the implementation rather than updating a clearly named constant. Additionally, the lack of a named constant makes the significance of this value less obvious to code reviewers and maintainers.

   Extract the hardcoded block time value into a named constant. Define a `BERACHAIN_BLOCK_TIME_SECONDS` constant at the module level and use it in the timestamp calculation:

   bera-reth/src/node/evm/config.rs::build_pending_env()

   ```
   const BERACHAIN_BLOCK_TIME_SECONDS: u64 = 2;

   impl BuildPendingEnv for BerachainNextBlockEnvAttributes {
       fn build_pending_env(parent: &SealedHeader) -> Self {
           Self {
               timestamp: parent.timestamp().saturating_add(BERACHAIN_BLOCK_TIME_SECONDS),
               // ... rest of implementation
           }
       }
   }
   ```

2. **Hardcoded System Address In PoL Execution**
   Related Asset(s): bera-reth/src/node/evm/executor.rs
   The PoL transaction execution in `execute_pol_transaction()` hardcodes `SYSTEM_ADDRESS` as the caller when invoking `transact_system_call()` instead of extracting the caller address from the PoL transaction itself. While

both the hardcoded value and the transaction's `from` field currently use `SYSTEM_ADDRESS`, this creates a maintainability issue where changes to the PoL transaction structure would not be reflected in the execution logic.

The PoL transaction is properly constructed with the caller address in `create_pol_transaction()`:

bera-reth/src/transaction/pol.rs::create_pol_transaction()

```
let pol_tx = PoLTx {
    chain_id: chain_spec.chain_id(),
    from: SYSTEM_ADDRESS,  //@audit caller address is set here
    to: chain_spec.pol_contract(),
    input: Bytes::from(calldata),
    // ... other fields
};
```

However, the executor ignores this field and hardcodes the same value:

bera-reth/src/node/evm/executor.rs::execute_pol_transaction()

```
// Execute as system call (maintains zero gas cost and unlimited gas)
match self.evm.transact_system_call(
    SYSTEM_ADDRESS,  //@audit hardcoded instead of using pol_tx.from
    pol_distributor_address,
    calldata.clone(),
) {
```

Extract the caller address from the PoL transaction's `from` field rather than hardcoding `SYSTEM_ADDRESS`. This would improve code maintainability and ensure consistency between transaction construction and execution logic.

3. **Misleading PoL Transaction Parameter Naming**

   Related Asset(s): bera-geth/core/types/tx_pol.go

   In `NewPoLTx()`, the parameter is named `distributionBlockNumber` but actually expects the previous block number (current block - 1), not the block number where the distribution occurs:

bera-geth/core/types/tx_pol.go::NewPoLTx()

```
func NewPoLTx(
    chainID *big.Int,
    distributorAddress common.Address,
    distributionBlockNumber *big.Int, //@audit misleading name
    gasLimit uint64,
    baseFee *big.Int,
    pubkey *common.Pubkey,
) (*Transaction, error) {
    // ...
    return NewTx(&PoLTx{
        // ...
        Nonce: distributionBlockNumber.Uint64(), // @audit actually previous block number
        // ...
    }), nil
}
```

In contrast, the `bera-reth` implementation uses clearer naming and handles the subtraction internally:

bera-reth/src/transaction/pol.rs::create_pol_transaction()

```
pub fn create_pol_transaction(
    chain_spec: Arc,
    prev_proposer_pubkey: BlsPublicKey,
    block_number: U256, // clearly the current block number
    base_fee: u64,
) -> Result {
    // ...
    let nonce_u256 = block_number - U256::from(1); // subtraction happens inside
    // ...
}
```

This inconsistency increases the likelihood of developer errors, as the parameter name suggests it should be the current block number but actually requires the previous block number.

Rename the parameter to `currentBlockNumber` and perform the subtraction inside the `NewPoLTx()` function to match the `bera-reth` implementation pattern.

4. **Redundant Proposer Public Key Validation**

   Related Asset(s): bera-reth/src/engine/rpc.rs

   The `validate_payload_v4_requirements()` function contains a redundant call to `validate_proposer_pubkey_prague1()`. The function first ensures that Prague1 is not active, then unnecessarily calls the proposer public key validation function.

bera-reth/src/engine/rpc.rs::validate_payload_v4_requirements()

```
fn validate_payload_v4_requirements(chain_spec: &ChainSpec, timestamp: u64) -> RpcResult<()> {
    if chain_spec.is_prague1_active_at_timestamp(timestamp) {
        return Err(EngineApiError::other(jsonrpsee_types::ErrorObject::owned(
            INVALID_PAYLOAD_ATTRIBUTES,
            "newPayloadV4P11 required for Prague1 fork, use newPayloadV4P11 instead",
            None::<()>,
        ))
        .into());
    }

    // Validate that no proposer pubkey is provided (should be None)
    // @audit This check is redundant
    validate_proposer_pubkey_prague1(chain_spec, timestamp, None).map_err(|error| {
        EngineApiError::other(jsonrpsee_types::ErrorObject::owned(
            INVALID_PAYLOAD_ATTRIBUTES,
            error.to_string(),
            None::<()>,
        ))
    })?;

    Ok(())
}
```

The `validate_proposer_pubkey_prague1()` function validates that proposer public keys are present after Prague1 activation and absent before Prague1 activation. Since `validate_payload_v4_requirements()` already ensures Prague1 is inactive and passes `None` for the proposer public key, this validation call will always succeed.

Remove the redundant `validate_proposer_pubkey_prague1()` call since the Prague1 activation check already ensures the validation would pass.

Recommendations

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

Resolution

The development team has implemented the following fixes for the issues raised above:

1. **Hardcoded Block Time Duration**: Fixed as recommended in bera-reth#123.
2. **Hardcoded System Address In PoL Execution**: Fixed as recommended in bera-reth#123.
3. **Misleading PoL Transaction Parameter Naming**: Fixed as recommended in `bera-geth` at commit ac2da12.

4. **Redundant Proposer Public Key Validation**: The redundant function has been replaced with a Prague1 timestamp check in bera-reth#63.

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.

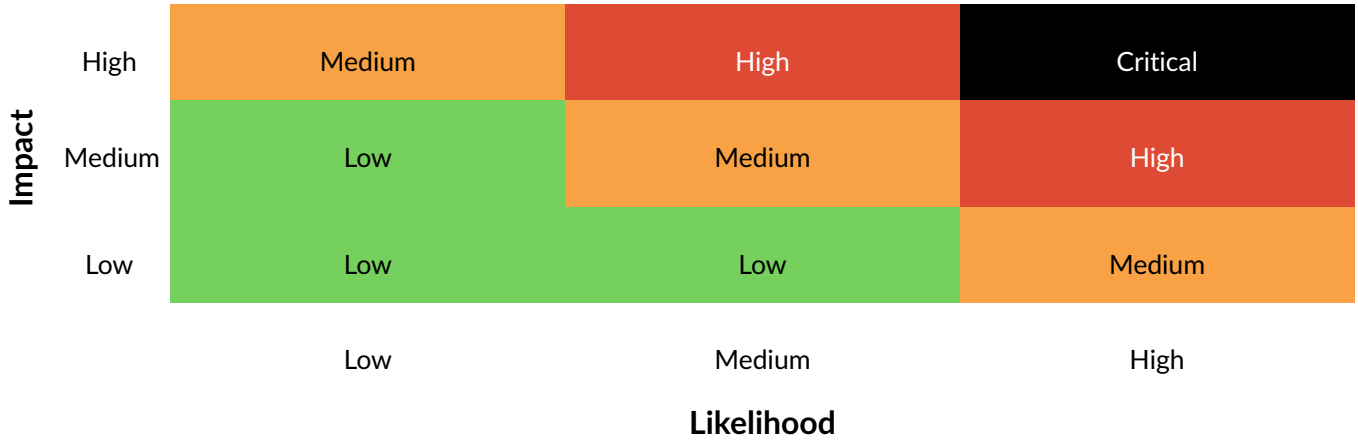

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.