OP Labs

Optimism Bedrock

Security Assessment Report

Version: 2.0

Contents

Introduction **2**
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary **4**
- Findings Summary . . . 4

Detailed Findings **5**

Summary of Findings **6**
- Panic When Unmarshalling Untrusted SSZ Data From P2P . . . 7
- Lack of Buffer Size Checks Before Using `FillBytes()` Function . . . 8
- Lack of Size Checks in `UnmarshalText()` Function . . . 9
- Unsafe Casting Leads to Integer Overflow . . . 10
- Insecure Order of Events in Verifying Unsafe L2 Blocks . . . 11
- Unsafe Use of Nonce to Identify Deposit Transactions . . . 12
- Private Key Stored Without Encryption . . . 14
- JWT Secret Stored Without Encryption . . . 15
- Unsafe Casting Used Throughout the Codebase . . . 16
- No Checks for Deposit Transaction Type When Calling `L1InfoDepositTxData()` . . . 17
- Inconsistent Handling of Deposit Transactions in Older Hard-Forks . . . 18
- Handling of Failed Deposit Transactions Could Lock ETH . . . 20
- Suboptimal Addition of JWT Authentication Header . . . 21
- Miscellaneous General Comments . . . 22

A Test Suite **25**
- Retesting . . . 27

B Vulnerability Severity Classification **28**

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Optimism's Rollup Node and Reference Optimistic Geth implementations, components of Optimism's *Bedrock* rollup architecture.

The review focused solely on the security aspects of the Golang implementation of the solution, 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 smart contract. 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 Optimism's Rollup Node and Reference Optimistic Geth code contained within the scope of the security review. These programs are henceforth abbreviated to *op-node* and *op-geth* respectively.

A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Appendix: 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*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite). The associated test code was provided alongside this report.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities identified within Optimism's Rollup Node and Reference Optimistic Geth.

Overview

Optimism is a Layer 2 (L2) Optimistic Rollup network designed to utilise the strong security guarantees of Ethereum while reducing its cost and latency.

The Optimism *Bedrock* architecture constitutes a significant redesign of the underlying rollup architecture.[1] The op-node and op-geth programs are key components of the Bedrock design.

Optimism's rollup node `op-node` is the component responsible for deriving the L2 chain from Layer 1 (L1) blocks (and their associated receipts). It functions as a L2 consensus-layer to the L2 execution layer, similar to how a beacon node (e.g. Prysm) utilises a L1 execution engine (e.g. Geth). The canonical chain of inputs is derived from L1 data, and rollup node drives an execution engine to extend the L2 chain with these inputs. Sequencers can also add their own inputs, to be submitted back to L1.

[1] Refer to `https://dev.optimism.io/introducing-optimism-bedrock/` for more information.

Optimism's fork of Go Ethereum `reference-optimistic-geth` (a.k.a. *op-geth*) provides the layer 2 execution engine. It has the core goal of being minimally different from the upstream Geth. The primary difference in Optimism's fork is introduction of the Deposit Transaction type, which is the mechanism for executing a transaction on L2 based on events and data taken from L1.

The op-geth block-building code is used by both the sequencer and verifier (a.k.a. validator) entities. When used to reconstruct blocks entirely based on L1 data, an extension to the regular Engine API fields is used to insert the right transactions and ignore the transaction pool.

Security Assessment Summary

This review was conducted on the files hosted on the optimism and reference-optimistic-geth repositories, and were assessed at commits:

- `optimism` : b708721, restricted to files within the `op-node` directory.
- `reference-optimistic-geth` (subsequently renamed to op-geth): 70b0248.

A subsequent round of testing targeted respective commits 055e4e7c and 68b97bc, and focused solely on verifying whether the previously identified issues had been resolved.

Note: *native Go and Go Ethereum libraries and dependencies were excluded from the primary focus of this assessment.*

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 Go runtime and Ethereum protocol.

To support this review, the testing team used the following automated testing tools:

- `golangci-lint`: `https://github.com/golangci/golangci-lint`
- `semgrep-go`: `https://github.com/dgryski/semgrep-go`
- native go fuzzing: `https://go.dev/doc/fuzz/`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 14 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 2 issues.
- Medium: 7 issues.
- Low: 1 issue.
- Informational: 3 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Optimism's Rollup Node and Reference Optimistic Geth implementations.

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.

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
OPB-01	Panic When Unmarshalling Untrusted SSZ Data From P2P	Critical	Resolved
OPB-02	Lack of Buffer Size Checks Before Using `FillBytes()` Function	High	Resolved
OPB-03	Lack of Size Checks in `UnmarshalText()` Function	High	Resolved
OPB-04	Unsafe Casting Leads to Integer Overflow	Medium	Resolved
OPB-05	Insecure Order of Events in Verifying Unsafe L2 Blocks	Medium	Resolved
OPB-06	Unsafe Use of Nonce to Identify Deposit Transactions	Medium	Resolved
OPB-07	Private Key Stored Without Encryption	Medium	Resolved
OPB-08	JWT Secret Stored Without Encryption	Medium	Closed
OPB-09	Unsafe Casting Used Throughout the Codebase	Medium	Closed
OPB-10	No Checks for Deposit Transaction Type When Calling `L1InfoDepositTxData()`	Medium	Resolved
OPB-11	Inconsistent Handling of Deposit Transactions in Older Hard-Forks	Low	Resolved
OPB-12	Handling of Failed Deposit Transactions Could Lock ETH	Informational	Closed
OPB-13	Suboptimal Addition of JWT Authentication Header	Informational	Resolved
OPB-14	Miscellaneous General Comments	Informational	Resolved

OPB-01	Panic When Unmarshalling Untrusted SSZ Data From P2P		
Asset	`op-node:eth/ssz.go`		
Status	**Resolved:** The issue has been resolved in PR 3360		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`(*ExectionPayload).UnmarshalSSZ()` fails to properly validate the `transactionsOffset` and `extraDataOffset` values, allowing a malicious actor to crash multiple op-nodes by gossiping a P2P message containing a specially crafted SSZ `ExecutionPayload`.

The function is missing checks to ensure that `transactionsOffset` is larger than `extraDataOffset`.

Consider the following `(*ExecutionPayload).UnmarshalSSZ()` function snippet below:

```
176   extraDataSize := transactionsOffset - extraDataOffset
      payload.ExtraData = make(BytesMax32, extraDataSize)
178   copy(payload.ExtraData, buf[extraDataOffset:transactionsOffset])
```

When `extraDataOffset > transactionsOffset` (which previous checks do not prevent), an unhandled "slice bounds out of range" panic is raised in the slicing operation at line [**178**].

The `uint32 extraDataSize` value also experiences a negative overflow, making it possible to allocate up to `4 GiB` at line [**177**] and exhaust the memory of some systems.

`(*ExecutionPayload).UnmarshalSSZ()` is executed on externally-accessible, untrusted data obtained from the P2P network at `p2p/gossip.go:221`. A malicious actor can trivially broadcast P2P messages that contain an `eth.ExecutionPayload` with its `transactionsOffset` smaller than `extraDataOffset`, triggering unhandled panics in multiple op-nodes when they unmarshal the data.

Note: this vulnerability would be triggered before validation that the execution payload had the Sequencer's valid signature, therefore allowing *anyone* to submit the malicious payload and trigger the crash — see OPB-05.

Refer to Appendix A for outputs from relevant proof-of-concept testing.

Recommendations

Implement checks to ensure `transactionsOffset` is larger than `extraDataOffset` before using these variables to create a slice.

Resolution

The issue has been resolved in PR 3360 by introducing offset checks in `op-node:etc/ssz.go` on line [**172**].

OPB-02	Lack of Buffer Size Checks Before Using `FillBytes()` Function		
Asset	`op-node:p2p/signer.go`, `op-node:rollup/derive/l1_block_info.go`, `op-node:rollup/derive/deposit_log.go`		
Status	**Resolved:** The issue has been resolved in PR 3620 and PR 3674		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There are no buffer size checks implemented before using `FillBytes()` function of `math/big` library.

According to `https://pkg.go.dev/math/big#Int.FillBytes` of `big.Int` and its `FilledBytes()` function - *"FillBytes() panics if the absolute value of X doesn't fit in buffer"*.

The `FillBytes()` is used on the following variables without sufficient validation of the buffer size, potentially causing unhandled panics if large absolute values are used:

- `chainID` in `op-node:p2p/signer.go`
- `info.BaseFee` in `op-node:rollup/derive/l1_block_info.go`
- `deposit.Mint` and `deposit.Value` in `op-node:rollup/derive/deposit_log.go`

Refer to Appendix A for outputs from relevant proof-of-concept testing.

Recommendations

Implement additional checks before using `FillBytes()` function to ensure the provided buffer is sufficiently large to fit required value.

Resolution

The issue has been resolved in PR 3620 and PR 3674 by introducing size checks on above values prior to calling `FillBytes()`.

OPB-03	Lack of Size Checks in `UnmarshalText()` Function		
Asset	`op-node:rollup/derive/params.go`		
Status	**Resolved:** The issue has been resolved since commit `21627e4` and PR 3359		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There are no size checks implemented in `(*ChannelID).UnmarshalText()` function to verify if text string being processed is of sufficient size.

```
if c := text[ChannelIDDataSize*2]; c != ':' {
  return fmt.Errorf("expected : separator in channel ID, but got %d", c)
}
```

If `text` is smaller than `ChannelIDDataSize*2`, the above code will trigger unhandled panic.

Recommendations

Implement checks to ensure supplied text string is of sufficient size to perform any data manipulation / extraction on.

Resolution

The vulnerable code has been removed in commit `21627e4` and PR 3359.

OPB-04	Unsafe Casting Leads to Integer Overflow		
Asset	`op-node:eth/ssz.go`		
Status	**Resolved:** The issue has been resolved in PR 3530		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Unsafe casting of `payload.ExtraData` length to `uint32` in `MarshalSSZ()` function may result in integer overflow, leading to unhandled panic.

On line [**102**] the following casting occurs in `copy()` function call:

```
copy(buf[offset:offset+uint32(len(payload.ExtraData))], payload.ExtraData[:])
```

Based on the snippet below from `op-node:eth/types.go`, type of `payload.ExtraData` is `[]byte` and, as such, its length can be bigger than `uint32`:

```
type BytesMax32 []byte
ExtraData     BytesMax32      `json:"extraData"`
```

When casting `len((payload.ExtraData))` to `uint32`, with sufficiently large `payload.ExtraData`, the result of `offset+uint32(len(payload.ExtraData))` will overflow. The final result will be "wrapped around", producing a smaller value than expected and subsequently triggering panic in the `copy()` function due to incorrect slice bounds.

Refer to Appendix A for outputs from relevant proof-of-concept testing.

Recommendations

Implement size checks before type casting to ensure the result will not overflow. Consider using `safecast` functions instead.

Verify values of variables before using them to create a slice.

Resolution

The issue has been resolved in PR 3530 by verifying size of `payload.ExtraData()` before casting.

OPB-05	Insecure Order of Events in Verifying Unsafe L2 Blocks		
Asset	`op-node:p2p/gossip.go`		
Status	**Resolved:** The issue has been resolved in PR 3361		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Sequencer's signature verifification is currently implemented as one of the last steps in the unsafe L2 block verification process.

Although computationally expensive, signature verification should be performed before unmarshalling execution payloads, which could be originating from untrusted and potentially malicious endpoints.

Recommendations

Ensure signature is verified before unmarshalling data to filter out potentially malicious payloads originating from untrusted endpoints.

Resolution

The issue has been resolved in PR 3361 by verifying signature before unmarshalling any data.

OPB-06	Unsafe Use of Nonce to Identify Deposit Transactions		
Asset	`op-geth:core/state_transition.go` & `core/types/deposit_tx.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Op-geth uses a "magic" transaction nonce value to identify deposit transactions within the state transition code. There are no validation steps preventing "normal" L2 transactions from having this nonce. As such, transactions originating from L2 that contain this nonce will be treated as deposit transactions within the state execution code.

Senders of these affected transactions do not have to pay gas fees, and other checks are bypassed that might result in problematic state execution.

The testing team did not identify any validation performed within `op-geth` or `op-node` that would prohibit non-deposit transactions from using this nonce value.

However, exploitation is only feasible when malicious sequencers or batch submitters are considered (with a non-standard transaction pool implementation); Hence the currently *low* likelihood rating.

Details

The Go Ethereum state transition code is executed on objects implementing the `Message` interface,[2] rather than a `types.Transaction`.[3] This is to allow state execution with input other than signed transactions, like to implement the `eth_call` and `eth_estimateGas` RPC endpoints. Because the `Message` interface abstracts away any underlying transaction type, `tx.Type() == DepositTxType` cannot be used to identify deposit transaction messages.

The op-geth design treats deposit transactions specially, as their gas was instead paid for on L1. To identify these deposit transaction messages, their nonce is reported as a large value that individual accounts are unlikely to reach.

```
const DepositsNonce uint64 = 0xffff_ffff_ffff_fffd
```

This magic value is then used to identify deposit transactions within the `StateTransition` code, like below:

```
func (st *StateTransition) preCheck() error {
  if st.msg.Nonce() == types.DepositsNonce {
    // No fee fields to check, no nonce to check, and no need to check if EOA (L1 already verified it for us)
    // Gas is free, but no refunds!
    st.initialGas = st.msg.Gas()
    st.gas += st.msg.Gas() // Add gas here in order to be able to execute calls.
    return nil
  }
  // ...
```

Because nonce validation is performed *by* the state transition code (at lines [**240-251**]) and bypassed for "deposit transactions", *there is no need for the sender's nonce to actually equal* `DepositsNonce`. Provided their transactions are included in a batch, a single sending account could repeatedly exploit this issue.

[2] Defined at `core/state_transition.go:67`.
[3] Defined at `core/types/transaction.go:53`.

Fortunately, the transaction pool (TxPool) implementation (used in op-geth for sequencer block production) appears to ensure only valid, incrementally increasing nonces are included in blocks. As such, a non-malicious sequencer would only include problematic transactions when the sending account actually has a current nonce of `DepositsNonce`.

Recommendations

Ensure deposit transactions can be uniquely identified within the `StateTransition` object's methods, to ensure gas and state logic specific to the `DepositTx` transactions is only applied to those transactions. Prefer to avoid relying on a trusted sequencer that proposes blocks containing transactions with valid nonces.

Add relevant tests to confirm expected behaviour.

Possible solutions appear to fall broadly into the following categories:

Explicitly Pass Type Information

One solution could involve adding another method to the `Message` interface, like `Type() uint64` or `IsDeposit() bool` (the latter appears more explicit).

As a caveat, changes to the interface may affect compatibility with third-party tools that would use op-geth as a library dependency. However, as op-geth already adds the `Mint()` and `RollupDataGas()` methods, this does not appear to pose any additional problems.

Nonce Validation

An alternative remediation could involve performing sufficient validation steps, to prohibit non-deposit transactions from using the `DepositsNonce` value. However, the testing team believe this to be a sub-optimal fix and may easily lead to other issues in the future; it does not resolve the underlying misuse of the `Nonce()` for other than its intended purpose. It also slightly breaks equivalence with the L1 EVM.

If the `DepositsNonce` mechanism were to be retained, it may also be necessary to implement a similar validation step in the `op-node`. At the least, care should be taken to ensure any validation in the TxPool (performed by the sequencer) should match the validation of batched transactions, to protect against the sequencer proposing invalid blocks.

Resolution

The issue has been resolved in PR 7, which exposes the type information via `IsDepositTx() bool` methods and replaces the magic nonce value.

OPB-07	Private Key Stored Without Encryption		
Asset	`op-node:p2p/signer.go`		
Status	**Resolved:** The issue has been resolved in PR 3547		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The batch submitter's private key is stored in cleartext.

The `LoadSignerSetup()` function uses `github.com/ethereum/go-ethereum/crypto` library and its `LoadECDSA()`, which loads private key from a file. There is no functionality decrypting key after loading it, suggesting that the batch submitter's private key is stored in cleartext.

Note: the testing team acknowledges this issue is known by the Optimism team and assumes there are plans to fix it, based on this comment found in the code '// TODO: load from encrypted keystore

Recommendations

Do not store sensitive information, such as private keys, in cleartext. Implement encryption for sensitive data at rest.

Resolution

The issue has been resolved in PR 3547 by reading the sequencer's private key from an environment variable rather than a file on disk.

OPB-08	JWT Secret Stored Without Encryption		
Asset	`op-node:service.go`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The JWT secret is stored in cleartext file.

There are no encryption mechanisms implemented in `NewL2EndpointConfig()` when saving secret to a file, and no decryption is in place when reading the file.

Recommendations

Do not store sensitive information, such as secret phrases or private keys, in cleartext. Implement encryption for sensitive data at rest.

Resolution

The issue has been risk accepted by the project team and no in-code mitigations were implemented.

The project team has advised:

"L1 Eth1/Eth2 clients do this the same way, and we have the same security model."

OPB-09	Unsafe Casting Used Throughout the Codebase		
Asset	`op-node:*`		
Status	**Closed:** The issue has been risk accepted by the project team and no mitigations were implemented.		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Unsafe type casting, such as `uint32()` or `uint64()`, is used throughout the codebase of the project.

Casting without bounds checking can cause integer overflows or underflows, leading to unexpected behaviour or unhandled crashes.

Recommendations

Implement bounds checks for all casting used in the project. Consider using the go-safecast library instead.

Resolution

The project team has advised:

"We reviewed all of the usages of 'unsafe' casting during the retreat, and didn't encounter any cases that looked insecure."

OPB-10	No Checks for Deposit Transaction Type When Calling `L1InfoDepositTxData()`		
Asset	`op-node:rollup/derive/channel_out.go`		
Status	**Resolved:** The issue has been resolved in PR 3548		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

It is assumed that the first transaction in the block is a deposit transaction, however, there are no checks implemented to verify it before calling `L1InfoDepositTxData()` function:

```
l1InfoTx := block.Transactions()[0]
l1Info, err := L1InfoDepositTxData(l1InfoTx.Data())
```

Usage of invalid data could lead to unexpected behaviour or unhandled panic.

Recommendations

Verify the type of block transaction before using it in the `L1InfoDepositTxData()` function, for example:

```
if l1InfoTx.Type() == types.DepositTxType {
  l1Info, err := L1InfoDepositTxData(l1InfoTx.Data())
}
```

Resolution

The issue has been resolved in PR 3548 by verifying transaction type before processing.

OPB-11	Inconsistent Handling of Deposit Transactions in Older Hard-Forks		
Asset	`op-geth:core/types/transaction_signing.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The transaction signers, responsible for validating and processing transaction signatures, handle deposit transactions (of `DepositTxType`) inconsistently. If signers other than `londonSigner` are ever used, they may cause unexpected behaviour and incorrectly validate deposit transactions.

Each implementation of the `Signer` interface performs transaction signature processing according to rules introduced by a particular Ethereum protocol upgrade.[4]

As these signers are not directly used as part of operating a node with the current devnet configuration, which has the `londonSigner` enabled from genesis,[5], this poses no identified risk to the Bedrock node software or the network.

Problems may otherwise occur if op-geth were used for processing transactions directly received from wallets, which can make use of the older signers. Any such contrived attack would be limited to impacting a user's own node.

Details

Op-geth introduces changes to the `londonSigner` to handle deposit transactions, which are not signed and do not have a signing hash. Other signers, which have not been modified, handle a transaction of `DepositTxType` differently. When passed the same transaction of `DepositTxType`, each signer behaves as follows:

`londonSigner`:

- `SignatureValues()` — returns error `"deposits do not have a signature"`.
- `Hash()` — panics.

`EIP2930Signer`:

- `SignatureValues()` — returns error `ErrTxTypeNotSupported`.
- `Hash()` — returns empty hash.

`EIP155Signer`:

- `SignatureValues()` — returns error `ErrTxTypeNotSupported`.
- `Hash()` — returns a non-empty hash (with a `chainId` input).

[4] A protocol upgrade, refer to `https://ethereum.org/en/history/` for further explanation.
For example, the `EIP155Signer` validates transactions that were accepted on Ethereum mainnet as of the Spurious Dragon update when EIP155 was introduced.

[5] Refer to the hardhat task at `optimism:packages/contracts-bedrock/tasks/genesis-l2.ts`, which sets `londonBlock: 0` at line [278].

`HomesteadSigner`:

- `SignatureValues()` — returns error `ErrTxTypeNotSupported`.
- `Hash()` — returns a different non-empty hash without error (with no `chainId` input).

`FrontierSigner`:
Same as `HomesteadSigner`.

Recommendations

Confirm whether an Optimism Bedrock chain could have a valid configuration where the London hard–fork is not immediately active at genesis.

If so, ensure consistent handling of `DepositTxType` transactions explicitly in the other signer implementations.

If the London hard-fork is required, consider explicitly enforcing this during configuration processing.

Resolution

The development team have confirmed that the design intends the London hard–fork to be enabled at genesis.

OPB-12	Handling of Failed Deposit Transactions Could Lock ETH
Asset	`op-geth:core/state_transition.go`
Status	**Closed:** See Resolution
Rating	Informational

Description

The current handling of deposit transactions may result in locked ETH when the transaction fails to execute and the L1 sender was a contract.

When a deposit transaction fails to execute, the sender's account balance is still credited with the `mint` value. This is an understandable design decision, to protect against ETH being indefinitely locked in the L1 deposit contract.

However, when the deposit's L1 sender is a contract, the sender on L2 (`CALLER`) is an alias. This alias will receive the mint funds but the sending contract may not be equipped to access or recover these funds (outside of a successful transaction).

Recommendations

Consider an alternative mechanism, in which failing deposit transactions initiate a withdrawal back to L1 (to allow recovery).

Alternatively, ensure documentation clearly warns bridge and contract developers to treat the deposit transactions with a non-zero mint value as effectively two independent state changes (where the transaction content may revert but the mint always succeeds).

So contracts interacting with the Optimism deposit bridge should be designed to recover any extra balance held by its L2 account alias.

Resolution

The development team intends to provide improved documentation to mitigate associated risks.

OPB-13	Suboptimal Addition of JWT Authentication Header
Asset	`op-geth:rpc/auth.go`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`(*JWTAuthProvider).AddAuthHeader()` does not correctly handle a scenario where the provided header already contains an entry with a key of `"Authorization"`. In this case, the function produces a malformed `http.Header` without returning an error.

`(*http.Header).Add()`, used at line [66], appends to any existing value associated with that key, resulting in an invalid `"Authorization"` header.

No other RPC client code currently sets an `"Authorization"` header. As such, this is not currently problematic and is deemed of *informational* severity.

Recommendations

Consider modifying `(*JWTAuthProvider).AddAuthHeader()` to appropriately handle the edge case where the provided `header` argument already contains an entry Authorization. This could involve either:

a) Returning an error if an `"Authorization"` entry already exists and has a non-empty value, or
b) Using `(*http.Header).Set()` to override the existing entry.

Add to the godoc comment describing this behaviour.

Resolution

After rebasing op-geth onto a more recent upstream codebase, this `AddAuthHeader()` method no longer exists.

The replacement intended for use by RPC clients (including op-node) is `NewJWTAuth()`, defined at `node/jwt_auth.go:33`. This uses `(*http.Header).Set()` and does not contain the same flaw.

OPB-14	Miscellaneous General Comments
Asset	`op-node:*` & `op-geth:*`
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 `Gas` value of 150_000** in `op-node:rollup/derive/l1_block_info.go`:
   Verify if the hardocded gas value is correct or could it be further optimised or calculated dynamically.
2. **`MaxReorgDepth` value hardcoded to 500** in `op-node:rollup/sync/start.go`:
   Verify if the hardcoded value is sufficient, 500 appears to be a narrow window of time considering fixed L2 block time of 2 seconds.
3. **Existence of `L1TrustRPC` flag:**
   Whilst we understand potential performance improvements of setting `L1TrustRPC` to `true`, no external RPC endpoints should be blindly trusted and verification should always be performed. Consider removal.
   Note, the default value is `false`.
4. **Unknown "magic" numbers:**
   - At `op-geth:core/types/transaction.go:345`, `(*Transaction).RollupDataGas()` contains the following unexplained number literal.
     ```
     onesGas := (ones + 68) * params.TxDataNonZeroGasEIP2028
     ```
     One can infer that this corresponds to some fixed, per-tx cost or profit fee but the reason for that specific number is unclear.
     Consider replacing the literal with a descriptively named `const` or config value, and include a comment explaining some reasoning for its use and value. This may also be worthwhile explaining in user-facing documentation.
5. **Numerous TODO comments to address:**
   We have observed numerous `TODO` comments throughout the code outlining outstanding design decisions and feature implementation.
   Go through all `TODO` comments to ensure all key design decisions have been made, verified and there are no outstanding items that could be considered critical.
6. **Misleading or incorrect comments:**
   At `miner/worker.go:1104`, we observe the following comment:
   ```
   // We use the eip155 signer regardless of the current hf.
   from, _ := types.Sender(work.signer, tx)
   ```
   However, this is incorrect and misleading. The signer used is instead chosen based on the current fork in `(*worker).makeEnv()` at line [769].
7. **Missing or insufficient godoc comments:**

- At `op-geth:rpc/client.go:189`, the godoc comment for `DialWithAuth()` is missing. It is preferable to clarify in the comment that no authentication is implemented for the stdio and IPC transports. This could otherwise lead to confusion.
- At `core/types/transaction_signing.go:226`, we recommend adding to the existing godoc comment for `(londonSigner).Hash()` to indicate that it panics if passed a `DepositTxType` (if this behaviour remains).

8. **Minor optimisations and style nitpicks:**

- Prefer the `%w` format flag to `%v` when using `fmt.Errorf()` to construct errors containing other error content. This allows errors to be wrapped with extra information while still making it possible to match the underlying error with `errors.Is()`.[6]
  Relevant in scope instances are as follows:
  - `op-geth:eth/catalyst/api.go:204`
  - `op-node:eth/ssz.go:181`
  - `op-node:eth/types.go:178`
  - `op-node:l1/receipts.go:36`
  - `op-node:l1/types.go:140,46,59,86,108,121,189`
  - `op-node:l2/util.go:51,59,75,105,113`
  - `op-node:node/node.go:115`
  - `op-node:p2p/config.go:159,164,168,172,196,200,213,217,223,246,257,307,325,349,375,393,397 401,405,412`
  - `op-node:p2p/discovery.go:114,119,150,157,161`
  - `op-node:p2p/gossip.go:192,325,331,354,358,362,374`
  - `op-node:p2p/host.go:54,59,63,66,71,76,81,87,149`
  - `op-node:p2p/node.go:59,74,79,91,132,137`
  - `op-node:p2p/rpc_server.go:359`
  - `op-node:p2p/signer.go:91`
  - `op-node:rollup/derive/attributes_queue.go:61`
  - `op-node:rollup/derive/batch_queue.go:71`
  - `op-node:rollup/derive/batches.go:19`
  - `op-node:rollup/derive/channel_bank.go:204`
  - `op-node:rollup/derive/channel_in_reader.go:111`
  - `op-node:rollup/derive/channel_out.go:136,147`
  - `op-node:rollup/derive/l1_block_info.go:123,128`
  - `op-node:rollup/derive/l1_retrieval.go:79,82`
  - `op-node:rollup/derive/params.go:56,63`
  - `op-node:rollup/derive/payload_util.go:28,35`
  - `op-node:rollup/derive/pipeline.go:142,158`
  - `op-node:rollup/driver/state.go:346`
  - `op-node:rollup/driver/step.go:50,53`
  - `op-node:service.go:39,44,49,54,104,129,135`

  These can be detected using the errorlint linter via golangci-lint.

[6] Refer to the errors package documentation for more info.

- At `op-geth:rpc/websocket.go:241`, in the statement `err = auth.AddAuthHeader(&dialHeader)`, `err` is an existing variable that is assigned. In this instance, `err` is captured as a closure from the surrounding scope and *might* complicate its garbage collection. Prefer creating a new variable with `err := auth.AddAuthHeader(&dialHeader)`.

9. **List of identified typos:**

   - `op-geth:core/beacon/types.go:39` — "the" should be "then".
   - `op-geth:miner/worker.go:1109` — "reply" should be "replay". This appears copied from upstream at line [**889**], which also contains the same typo.

Recommendations

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

Resolution

The comments above have been acknowledged by the project team and relevant changes actioned where appropriate.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and were provided along with this document. The following were run from the `optimism/op-node` directory and output is given below.

The following output is associated with OPB-01.

```
$ go test -run=TestUnmarshalSSZ ./eth
--- FAIL: TestUnmarshalSSZ (0.03s)
panic: runtime error: slice bounds out of range [508:11] [recovered]
panic: runtime error: slice bounds out of range [508:11]

goroutine 6 [running]:
testing.tRunner.func1.2({0x7d1540, 0xc0000a0000})
/usr/local/go/src/testing/testing.go:1389 +0x24e
testing.tRunner.func1()
/usr/local/go/src/testing/testing.go:1392 +0x39f
panic({0x7d1540, 0xc0000a0000})
/usr/local/go/src/runtime/panic.go:838 +0x207
github.com/ethereum-optimism/optimism/op-node/eth.(*ExecutionPayload).UnmarshalSSZ(0xc00005bd38, 0x1fc, {0x888900, 0xc00018af90})
<optimism_root>/op-node/eth/ssz.go:178 +0x8af
github.com/ethereum-optimism/optimism/op-node/eth.TestUnmarshalSSZ(0x0?)
<optimism_root>/op-node/eth/sigp_ssz_test.go:79 +0xdf
testing.tRunner(0xc000115ba0, 0x824820)
/usr/local/go/src/testing/testing.go:1439 +0x102
created by testing.(*T).Run
/usr/local/go/src/testing/testing.go:1486 +0x35f
exit status 2
FAIL  github.com/ethereum-optimism/optimism/op-node/eth  0.040s
```

The following output is associated with OPB-02.

```
$ go test -run=TestSigningHash ./p2p
--- FAIL: TestSigningHash (0.00s)
panic: math/big: buffer too small to fit value [recovered]
  panic: math/big: buffer too small to fit value

goroutine 29 [running]:
testing.tRunner.func1.2({0xecd960, 0x145d770})
  /usr/local/go/src/testing/testing.go:1389 +0x24e
testing.tRunner.func1()
  /usr/local/go/src/testing/testing.go:1392 +0x39f
panic({0xecd960, 0x145d770})
  /usr/local/go/src/runtime/panic.go:838 +0x207
math/big.nat.bytes(...)
  /usr/local/go/src/math/big/nat.go:1166
math/big.(*Int).FillBytes(0xc000074f00, {0xc000164320, 0x20, 0x40})
  /usr/local/go/src/math/big/int.go:466 +0xf8
github.com/ethereum-optimism/optimism/op-node/p2p.SigningHash({0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...}, ...)
  <optimism_root>/op-node/p2p/signer.go:30 +0xa7
github.com/ethereum-optimism/optimism/op-node/p2p.TestSigningHash(0x409c99?)
  <optimism_root>/op-node/p2p/signer_test.go:16 +0x11b
testing.tRunner(0xc000103040, 0x13488a8)
  /usr/local/go/src/testing/testing.go:1439 +0x102
created by testing.(*T).Run
  /usr/local/go/src/testing/testing.go:1486 +0x35f
exit status 2
FAIL  github.com/ethereum-optimism/optimism/op-node/p2p  0.021s
```

The following output is associated with OPB-02.

```
$ go test -run=TestMarshalBinary ./rollup/derive
--- FAIL: TestMarshalBinary (0.00s)
panic: math/big: buffer too small to fit value [recovered]
  panic: math/big: buffer too small to fit value

goroutine 16 [running]:
testing.tRunner.func1.2({0x909ee0, 0xaa9440})
  /usr/local/go/src/testing/testing.go:1389 +0x24e
testing.tRunner.func1()
  /usr/local/go/src/testing/testing.go:1392 +0x39f
panic({0x909ee0, 0xaa9440})
  /usr/local/go/src/runtime/panic.go:838 +0x207
math/big.nat.bytes(...)
  /usr/local/go/src/math/big/nat.go:1166
math/big.(*Int).FillBytes(0xc000067ec0, {0xc0001961a4, 0x20, 0x60})
  /usr/local/go/src/math/big/int.go:466 +0xf8
github.com/ethereum-optimism/optimism/op-node/rollup/derive.(*L1BlockInfo).MarshalBinary(0xc0000c7f20)
  <optimism_root>/op-node/rollup/derive/l1_block_info.go:44 +0x94
github.com/ethereum-optimism/optimism/op-node/rollup/derive.TestMarshalBinary(0x0?)
  <optimism_root>/op-node/rollup/derive/derive_test.go:25 +0x13c
testing.tRunner(0xc0001e1860, 0xa189a0)
  /usr/local/go/src/testing/testing.go:1439 +0x102
created by testing.(*T).Run
  /usr/local/go/src/testing/testing.go:1486 +0x35f
exit status 2
FAIL  github.com/ethereum-optimism/optimism/op-node/rollup/derive  0.013s
```

The following output is associated with OPB-02.

```
$ go test -run=TestMarshalDepositLogEvent ./rollup/derive
--- FAIL: TestMarshalDepositLogEvent (0.00s)
panic: math/big: buffer too small to fit value [recovered]
  panic: math/big: buffer too small to fit value

goroutine 16 [running]:
testing.tRunner.func1.2({0x909ee0, 0xaa9520})
  /usr/local/go/src/testing/testing.go:1389 +0x24e
testing.tRunner.func1()
  /usr/local/go/src/testing/testing.go:1392 +0x39f
panic({0x909ee0, 0xaa9520})
  /usr/local/go/src/runtime/panic.go:838 +0x207
math/big.nat.bytes(...)
  /usr/local/go/src/math/big/nat.go:1166
math/big.(*Int).FillBytes(0xc0001ca6a0, {0xc0002e8600, 0x20, 0x25e1})
  /usr/local/go/src/math/big/int.go:466 +0xf8
github.com/ethereum-optimism/optimism/op-node/rollup/derive.marshalDepositVersion0(0xc0001b75e0)
  <optimism_root>/op-node/rollup/derive/deposit_log.go:191 +0x7a
github.com/ethereum-optimism/optimism/op-node/rollup/derive.MarshalDepositLogEvent({0xde, 0xad, 0xbe, 0xef, 0xde, 0xad, 0xbe, 0xef,
      ↪  0xde, 0xad, ...}, ...)
  <optimism_root>/op-node/rollup/derive/deposit_log.go:157 +0x1af
github.com/ethereum-optimism/optimism/op-node/rollup/derive.TestMarshalDepositLogEvent(0x0?)
  <optimism_root>/op-node/rollup/derive/derive_test.go:44 +0x24b
testing.tRunner(0xc0001e16c0, 0xa189c8)
  /usr/local/go/src/testing/testing.go:1439 +0x102
created by testing.(*T).Run
  /usr/local/go/src/testing/testing.go:1486 +0x35f
exit status 2
FAIL  github.com/ethereum-optimism/optimism/op-node/rollup/derive  0.013s
```

The following output is associated with OPB-04.

```
$ go test -run TestMarshalSSZ ./eth
--- FAIL: TestMarshalSSZ (0.03s)
panic: runtime error: slice bounds out of range [508:4] [recovered]
  panic: runtime error: slice bounds out of range [508:4]

goroutine 6 [running]:
testing.tRunner.func1.2({0x7ca160, 0xc0000182b8})
  /usr/local/go/src/testing/testing.go:1389 +0x24e
testing.tRunner.func1()
  /usr/local/go/src/testing/testing.go:1392 +0x39f
panic({0x7ca160, 0xc0000182b8})
  /usr/local/go/src/runtime/panic.go:838 +0x207
github.com/ethereum-optimism/optimism/op-node/eth.(*ExecutionPayload).MarshalSSZ(0xc000059d38, {0x880220, 0xc10019c030})
  <optimism_root>/op-node/eth/ssz.go:102 +0x705
github.com/ethereum-optimism/optimism/op-node/eth.TestMarshalSSZ(0xc000095a00)
  <optimism_root>/op-node/eth/sigp_ssz_test.go:53 +0x3c5
testing.tRunner(0xc000095a00, 0x81cb48)
  /usr/local/go/src/testing/testing.go:1439 +0x102
created by testing.(*T).Run
  /usr/local/go/src/testing/testing.go:1486 +0x35f
exit status 2
FAIL  github.com/ethereum-optimism/optimism/op-node/eth  0.047s
```

Retesting

These tests were updated and re-run as part of retesting activities. All implemented tests subsequently passed.

Appendix B 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.

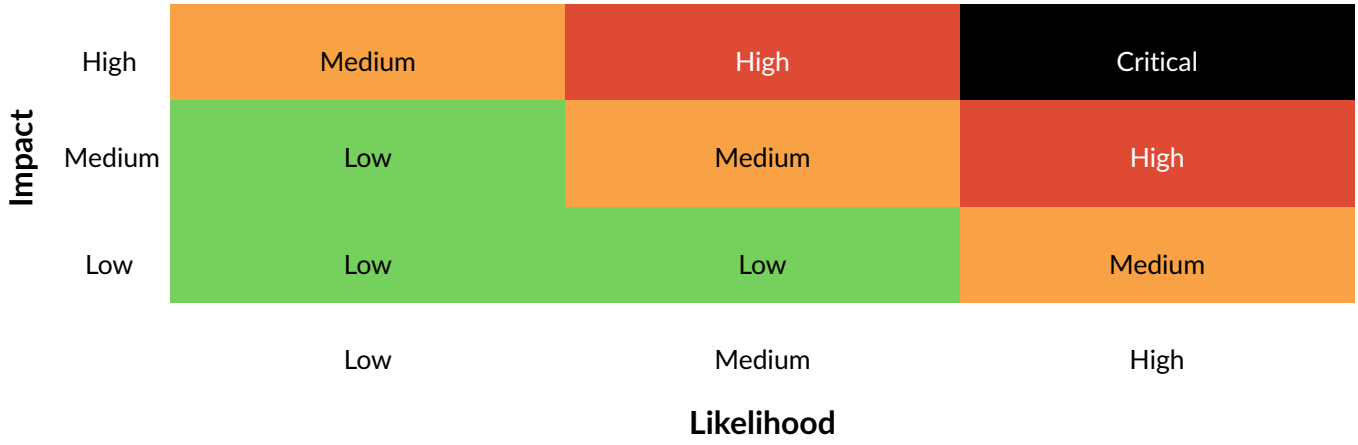

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References