Security Review Report
NM-0411-0491 Rollup Boost

NETHERMIND
SECURITY

(June 9, 2025)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 3

4 Summary of Security Implications to Permissioneless Builders 3

5 System Overview 4
5.1 Rollup Boost: Proposer-Builder Separation in L2 . . . 4
5.2 Core System Workflow of Rollup Boost . . . 4
5.3 HTTP Proxy Layer and Multiplexing Logic in `proxy.rs` . . . 5
5.4 Main Service Logic: The Role of `server.rs` . . . 5
5.5 RPC Module (`rpc.rs`) . . . 6

6 Risk Rating Methodology 8

7 Issues 9
7.1 [Medium] JWT unencrypted in HTTP connections . . . 9
7.2 [Medium] Server listening on all interfaces by default, without authentication . . . 9
7.3 [Low] Debug API authentication . . . 9
7.4 [Low] JWT tokens timestamped 60 seconds into the future . . . 9
7.5 [Low] Possible execution_mode inconsistency in `get_payload` . . . 10
7.6 [Low] Proxy does not implement rate-limiting . . . 10
7.7 [Low] Use of `sync::Cache` in an async context . . . 10
7.8 [Info] CI GitHub Action actions-rs unmaintained . . . 11
7.9 [Info] Code inside the `is_fallback_enabled()` branch may not be reachable . . . 12
7.10 [Info] Incoming requests have a large size limit . . . 13
7.11 [Info] `boost_sync` flag can cause discrepancies between builder and L2 op-geth . . . 14
7.12 [Info] getPayload requests to builder when FCU `no_tx_pool` is true . . . 15
7.13 [Best Practice] Concurrency in server . . . 15
7.14 [Best Practice] Different conditions for cache insertion/eviction . . . 15
7.15 [Best Practice] Do not clone payload in server|fcu_v3 . . . 16
7.16 [Best Practice] Silent failures of `tokio::spawn` . . . 16
7.17 [Best Practice] Using expect and unwrap in the code . . . 16
7.18 [Best Practice] CI use cargo deny . . . 16
7.19 [Best Practice] Do not clone payload in rpc|new_payload_v3 . . . 17
7.20 [Best Practice] Logic of forwarding requests should be improved . . . 17

8 Security Implications of Using Permissioneless Builders 18
8.1 MEV extraction attack like the one performed against Mev-Boost . . . 18
8.2 Do not blindly default to builder block . . . 18
8.3 Slow down (or even disable) *op-geth* with extremely large block from builder . . . 18

9 Documentation Evaluation 19
9.1 The code for `get_payload` is not easy to reason about . . . 19

10 Test Suite Evaluation 22
10.1 Code Coverage Analysis . . . 22
10.2 Tests Output . . . 23

11 About Nethermind 24

1 Executive Summary

This document presents the security review performed by Nethermind Security for Rollup Boost. Rollup Boost is a specialized sidecar service designed to enhance the flexibility and modularity of block production for Optimism Stack chains. It enables external block builders to construct blocks independently of the Sequencer's Execution Engine, allowing for more customizable transaction sequencing without modifying the core Optimism protocol.

The audited code comprises 2,634 lines of code written in Rust language. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. Fig. 1 summarizes the the identified issues into four severity levels. **In this report, we present** 20 points of attention: two are classified as `Medium`, five are classified as `Low`, and 13 are classified as `Informational` or `Best Practice`. In addition, we also discuss **security implications** to permissioneless builders in Section 4.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues and Section 4 summarizes security implications in the context of permissioneless builders. Section 5 presents the system overview. Section 6 discusses the risk rating methodology. Section 7 details the issues and Section 8 discuss the security implications of using permissioneless builders. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, and automated tests. Section 10 concludes the document.

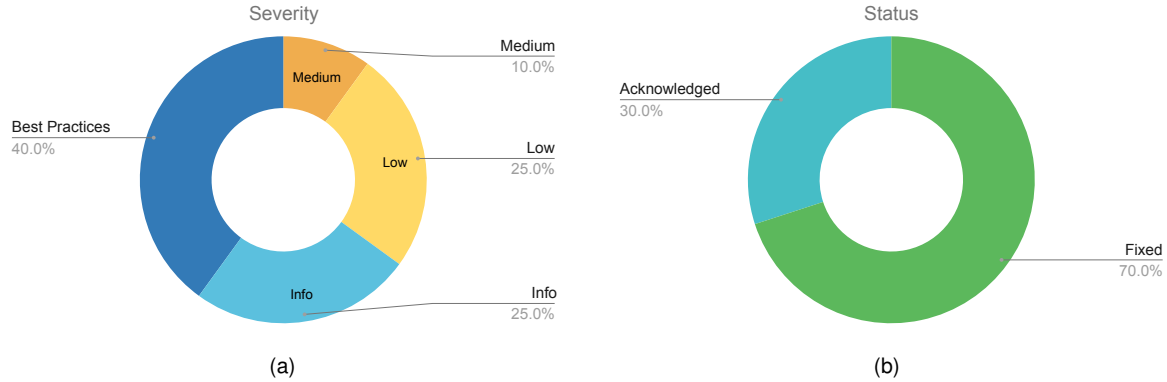

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (5), **Undetermined** (0), **Informational** (5), **Best Practices** (8). **Distribution of status: Fixed** (14), **Acknowledged** (6), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 11, 2025
Response from Client	Regular responses during audit engagement
Final Report	June 09, 2025
Repository	rollup-boost
Commit Audit	ea0ec43
Final Commit	71c4482
Documentation	README file and docs folder
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/server.rs	886	33	3.72%	122	11
2	src/proxy.rs	605	26	4.30%	108	708
3	src/client/rpc.rs	403	19	12.5%	50	128
4	src/cli.rs	176	22	6.77%	47	144
5	src/tracing.rs	148	10	7.32%	20	167
6	src/debug_api.rs	123	9	1.20%	36	34
7	src/client/http.rs	83	1	10.80%	15	10
8	src/client/auth.rs	65	7	1.70%	10	79
9	src/metrics.rs	59	1	1.70%	10	20
9	src/health.rs	57	1	1.75%	9	20
9	src/lib.rs	18	0	0.0%	8	20
9	src/bin/main.rs	8	0	0.0%	2	20
9	src/client/mod.rs	3	0	0.0%	0	20
	Total	**2,634**	**129**	**4.21%**	**437**	**1301**

3 Summary of Issues

	Finding	Severity	Update
1	JWT unencrypted in HTTP connections	Medium	Acknowledged
2	Server listening on all interfaces by default, without authentication	Medium	Fixed
3	Debug API authentication	Low	Acknowledged
4	JWT tokens timestamped 60 seconds into the future	Low	Fixed
5	Possible execution_mode inconsistency in `get_payload`	Low	Fixed
6	Proxy does not implement rate-limiting	Low	Acknowledged
7	Use of `sync::Cache` in an async context	Low	Fixed
8	CI GitHub Action actions-rs unmaintained	Info	Fixed
9	Code inside the `is_fallback_enabled()` branch may not be reachable	Info	Fixed
10	Incoming requests have a large size limit	Info	Fixed
11	`boost_sync` flag can cause discrepancies between builder and L2 op-geth	Info	Fixed
12	getPayload requests to builder when FCU `no_tx_pool` is true	Info	Fixed
13	Concurrency in server	Best Practices	Acknowledged
14	Different conditions for cache insertion/eviction	Best Practices	Fixed
15	Do not clone payload in server|fcu_v3	Best Practices	Fixed
16	Silent failures of `tokio::spawn`	Best Practices	Acknowledged
17	Using expect and unwrap in the code	Best Practices	Fixed
18	CI use cargo deny	Best Practices	Fixed
19	Do not clone payload in rpc|new_payload_v3	Best Practices	Fixed
20	Logic of forwarding requests should be improved	Best Practices	Acknowledged

4 Summary of Security Implications to Permissioneless Builders

	Description
1	Do not blindly default to builder block
2	Slow down (or even disable) op-geth with extremely large block from builder
3	MEV extraction attack like the one performed against Mev-Boost

5 System Overview

This section provides an overview of Rollup Boost at the initial audited commit `ea0ec43`, highlighting its architecture, components, and functionalities. **It is important to note that significant changes have occurred in the core logic during the resolution of identified issues. The final commit `71c4482` reflects these modifications, presenting the current state of the system post-review.**

5.1 Rollup Boost: Proposer-Builder Separation in L2

Ethereum Layer 1 has adopted a model called **Proposer-Builder Separation (PBS)**, where the block proposer delegates construction to an external builder that can optimize transaction ordering and MEV strategies. This separation, facilitated by tools such as `mev-boost`, enables modular and competitive block production.

Rollup Boost brings this PBS model to Layer 2 by decoupling the sequencer's execution logic from block construction, allowing external builders to generate blocks while maintaining protocol compatibility. This modularity opens up opportunities for experimentation, performance improvements, and greater decentralization in the OP Stack ecosystem. More information on the design here.

5.2 Core System Workflow of Rollup Boost

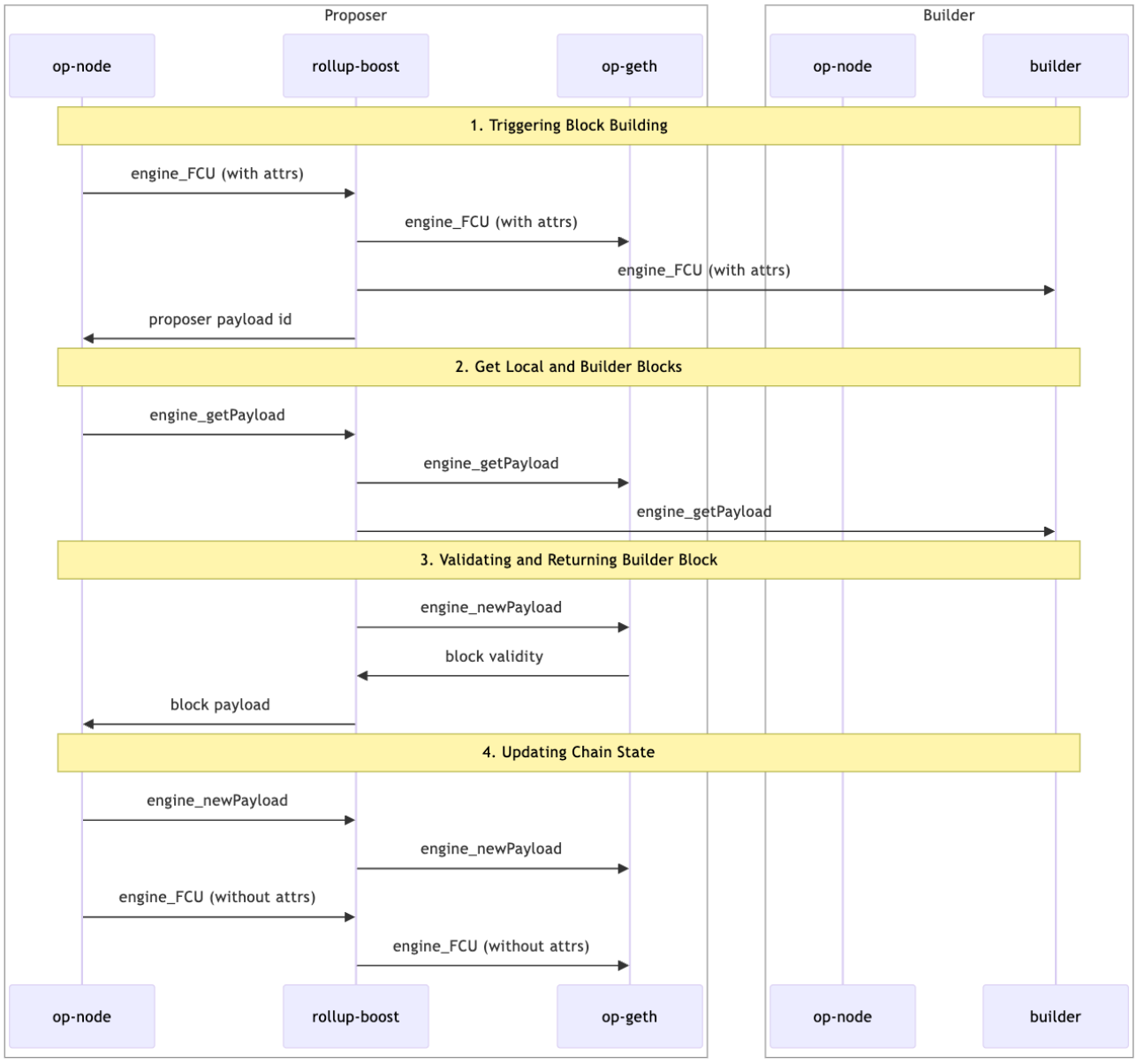

Fig. 2: Sequence diagram extracted from Core system workflow of Rollup Boost

The sequence diagram in Fig. 2 is derived from the Rollup Boost GitHub repository. Basically, it serves as a sidecar component for Optimism Stack chains, enabling external block production by introducing a separation between the *proposer* and *builder* roles. This design allows for greater flexibility in transaction sequencing and block construction. The core workflow operates as follows:

a. **Initiating Block Building**: When the `op-node` initiates block building, it sends an `engine_forkchoiceUpdated` (FCU) call with payload attributes to Rollup Boost. Rollup Boost forwards this call to both the local `op-geth` and the external builder's execution engine.

b. **Fetching Payloads**: Upon receiving an `engine_getPayload` request from the `op-node`, Rollup Boost concurrently requests payloads from both the local `op-geth` and the external builder.

c. **Validating and Selecting Payloads**: Once the builder's payload is received, Rollup Boost validates it by sending an `engine_newPayload` call to the local `op-geth`. If the payload is valid, it is returned to the `op-node`; otherwise, the fallback payload from the local `op-geth` is used.

d. **Updating Chain State**: The `op-node` proceeds to update the chain state by sending an `engine_newPayload` call and another `engine_forkchoiceUpdated` call without payload attributes to Rollup Boost, which forwards these calls to the local `op-geth`.

5.3 HTTP Proxy Layer and Multiplexing Logic in `proxy.rs`

The `proxy.rs` implements a proxy layer using the Tower middleware framework to intercept and route JSON-RPC HTTP requests based on their method. It is the core component responsible for determining whether a request should be:

- Sent to the rollup's local execution engine (`l2_client`),
- Sent to the external builder (`builder_client`),
- Or sent to both, in a *multiplexed* fashion.

(A). Core Components

`ProxyLayer` - A middleware layer that sets up `HttpClient` instances for both the local execution engine and the external builder. It wraps a service (`inner`) and adds health checking via `HealthLayer`.

`ProxyService` - The service implementation that inspects incoming HTTP JSON-RPC requests, reads the method, and applies routing logic. It clones and spawns requests asynchronously when needed.

(B). Routing Logic

The constant string arrays `MULTIPLEX_METHODS` and `FORWARD_REQUESTS` determine how each JSON-RPC method is handled:

- If a method starts with any prefix in `MULTIPLEX_METHODS`, it may be multiplexed (sent to both builder and L2).
- If it's also in `FORWARD_REQUESTS`, it's sent to both the builder and L2.
- Otherwise, it's only handled by the L2 or passed to the `inner` service.

```
if MULTIPLEX_METHODS.iter().any(|&m| method.starts_with(m)) {
    if FORWARD_REQUESTS.contains(&method.as_str()) {
        // Asynchronously forward to builder
        // Synchronously forward to L2
    } else {
        // Handled by the inner server service (proxied to the L2 and/or builder execution engine)
    }
} else {
    // Directly forward to L2 engine via HttpClient
}
```

5.4 Main Service Logic: The Role of `server.rs`

The `server.rs` file in the Rollup Boost repository serves as the main entry point for the sidecar service, orchestrating interactions between `op-node`, the local execution client (`op-geth`), and external block builders. It implements the Engine API methods—`engine_forkchoiceUpdated`, `engine_getPayload`, and `engine_newPayload`—to facilitate modular block production in the OP Stack.

(A). Core Responsibilities

The key responsibilities can be summarized in:

- **API Routing and Handling:** Defines HTTP endpoints corresponding to Engine API methods, parses incoming requests from the `op-node`, and routes them to the appropriate handlers.
- **Payload Management:** Coordinates the retrieval and validation of block payloads from both the local `op-geth` and external builders, ensuring selection of the most valid and optimal block.
- **State Coordination:** Maintains internal state to track block building processes, map proposer and builder payload IDs, and manage fallback strategies in case of builder failure.

(B). Core Functions

- `fork_choice_updated_v3`: Processes incoming `forkchoiceUpdatedV3` requests, starts the block building process, and manages payload ID mappings.
- `get_payload`: Handles payload requests (`getPayloadV3` and `getPayloadV4`) by fetching the built block from the local or external builder (preferred) and returning it to the `op-node`.
- `new_payload`: Forwards incoming block payloads (`newPayloadV3` and `newPayloadV4`) to the local `op-geth` for validation and returns the result to the caller.

(C). Execution Mode

Rollup Boost offers configurable execution modes that determine how it interacts with external block builders. These modes can be adjusted at runtime via the Debug API, allowing operators to tailor the behavior of Rollup Boost without restarting the service.

(1). *Available Execution Modes* - The functions below are called to determine the current execution mode and decide whether to forward the request to the external builder or rely exclusively on local engine.

```rust
pub enum ExecutionMode {
    Enabled,
    DryRun,
    Disabled,
    Fallback,
}

impl ExecutionMode {
    fn is_get_payload_enabled(&self) -> bool {
        matches!(self, ExecutionMode::Enabled)
    }

    fn is_disabled(&self) -> bool {
        matches!(self, ExecutionMode::Disabled)
    }

    fn is_fallback_enabled(&self) -> bool {
        matches!(self, ExecutionMode::Fallback)
    }
}
```

- **Enabled**: Rollup Boost forwards all Engine API calls to the external builder. This mode fully integrates the builder into the block production process.
- **DryRun**: All Engine API calls are sent to the builder, except for `engine_getPayload`.
- **Disabled**: No Engine API calls are sent to the builder.
- **Fallback**: Do not use blocks built by the builder. Only blocks from the local `op-geth` are used.

(2). *Fallback Mechanism* - When operating in `Enabled` mode, Rollup Boost attempts to retrieve and validate a payload from the external builder. If this payload is deemed invalid, typically by failing the `engine_newPayload` validation against the local `op-geth`, Rollup Boost falls back to a locally built payload. This fallback ensures that:

- The network remains live and responsive even if the external builder is unavailable or returns bad data.
- The `op-node` always receives a valid payload, regardless of builder reliability.

5.5 RPC Module (`rpc.rs`)

The `rpc.rs` module in Rollup Boost serves as the core interface for forwarding and handling Engine API calls, ensuring seamless integration and coordination among these components.

- The `RpcClient` struct defines the client-side interface for interacting with an execution layer node's Engine API in Rollup Boost. It encapsulates the logic required to make authenticated JSON-RPC requests, relying on a JWT-secured `auth_client` to communicate with the node.
- The `auth_rpc` field specifies the URI of the target RPC server, ensuring that the client knows where to route its requests.
- The `payload_source` field identifies the origin of the payloads, whether from the local execution engine or an external builder, enabling Rollup Boost to distinguish between different sources during block construction and validation.

```rust
pub struct RpcClient {
    auth_client: HttpClient<AuthClientService<HttpBackend>>,
    auth_rpc: Uri,
    payload_source: PayloadSource,
}
```

The module forwards specific Engine API calls, such as `engine_forkchoiceUpdatedV3`, `engine_getPayloadV3`, and `engine_newPayloadV3`, to both the local execution client and external builders. This multiplexing allows for parallel block building and validation processes.

Ⓐ. Functionality Overview

The `rpc.rs` module defines the structure and behavior of the RPC client used by Rollup Boost. It establishes connections to the local execution client and external builders using provided URLs and authentication tokens.

6 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

7 Issues

7.1 [Medium] JWT unencrypted in HTTP connections

File(s): `src/client/http.rs`

Description: Connections to the L2 and builder nodes are currently allowed to be through HTTP (`http://` URL) or HTTPS (`https://` URL). Assuming the user sets up a HTTP address for a client, and, given that communication over HTTP is unencrypted, the JSON Web Tokens employed for authentication will be also unencrypted. Though with a validity period of only 60 seconds, in non-local setups (e.g., remote builder), an eavesdropper can easily sniff rollup-boost -> builder packets and gather the JWTs in use. With a hold of these, the eavesdropper can then impersonate the rollup-boost.

Recommendation(s): Enforce HTTPS-only connections through the setter `ConnectionBuilder::https_only()`, or warn the user on the use of HTTP in (possible) non-local connections.

Status: Acknowledged

Update from the client: This mirrors the behavior of all major execution clients, which also allow JSON-RPC over HTTP even when JWT authentication is enabled. Rollup Boost follows the same model as existing Ethereum execution clients. HTTP should only used in trusted or local environments.

7.2 [Medium] Server listening on all interfaces by default, without authentication

File(s): `src/cli.rs`

Description: While configurable by the user, the default `0.0.0.0` makes the server accept incoming connections from any interface, possibly allowing an actor to inject malicious packets which are then signed (using rollup-boost's L2/builder JWT secrets) and forwarded to the L2 and builder nodes. The exposure of this issue is aggravated given that there is no use of an authentication mechanism for incoming messages.

Recommendation(s): Consider changing the default listening IP to `localhost` / `127.0.0.1`.

Status: Fixed

Update from the client: Fixed in d3ab9ac

7.3 [Low] Debug API authentication

File(s): `src/debug_api.rs`

Description: The debug API allows runtime modification of critical server state (`ExecutionMode`) and it doesn't seem to have any application-level authentication, security relies only on port accessibility or firewall.

Recommendation(s): Implement mandatory authentication.

Status: Acknowledged

Update from the client: Rollup Boost is not intended to be publicly exposed. Therefore the authentication of the debug API is unnecessary. It's worth mentioning that the same solution is used in op-node's admin RPC.

7.4 [Low] JWT tokens timestamped 60 seconds into the future

File(s): `src/client/auth.rs`

Description: JWT claims used in the engine API are mandated to be valid for +- 60 seconds. However, the current `secret_to_bearer_header()` function creates JWTs issued 60 seconds into the future. This provides a possible eavesdropper to use the JWT tokens for double their intended duration. Moreover, if the system times of the rollup-boost and the receiving node (L2 or builder) are skewed apart more than the latency between these, the receiving node will block the rollup-boost requests due to a perceived `iat` being more than 60 seconds into the future.

Recommendation(s): Remove the addition of the 60 seconds duration from the `iat` calculation. Use only current time.

Status: Fixed

Update from the client: Fixed in 4e0e83b

7.5 [Low] Possible execution_mode inconsistency in `get_payload`

File(s): server.rs

Description: When `get_payload` is called the flow is:

1. Try to get payload from both builder and L2 execution client;

1.1. To decide whether payload should be fetched from external builder, the `execution_mode` is fetched (link) 1.2. The way it is fetched is that the mutex is locked and value is copied 2. Once payload is received, *process it*. 2.1. During payload *processing* the `execution_mode` is fetched yet again by locking the mutex and reading the value.

The issue is that between points 1 & 2, *some time* can pass, and the value of `execution_mode` might be changed in-between. What can happen is:

1. `execution_mode` is set to `Enabled`, which means the builder will start building the block;
2. Actor who has access to `debug_api` can change `execution_mode` to `Disabled`;
3. Builder will return `Ok(Some(payload))`;
4. Even though execution is set to `Disabled` we will send the payload from the builder instead from the L2 Execution Client. This is because:;

```
        let (payload, context) = match (builder_payload, l2_payload) {
            (Ok(Some(builder)), Ok(l2_payload)) => {
                if self.execution_mode().is_fallback_enabled() {
                    // Default to op-geth's payload
                    Ok((l2_payload, PayloadSource::L2))
                } else {
                    Ok((builder, PayloadSource::Builder))
                }
            }
            (_, Ok(l2)) => Ok((l2, PayloadSource::L2)),
            (_, Err(e)) => Err(e),
        }?;
```

We will end up in a branch: `(Ok(Some(builder)), Ok(l2_payload))` but `is_fallback_enabled` will be false.

Recommendation(s): Read `execution_mode` once, at the beginning of `get_payload`.

Status: Fixed

Update from the client: Fixed in PR 198

7.6 [Low] Proxy does not implement rate-limiting

File(s): `src/proxy.rs`

Description: Rate-limiting helps protect systems from abuse, brute-force attacks, and resource overuse by restricting how often a user can make requests. While the rollup-boost main server implements some rate-limiting through `jsonrpsee::Server`, the proxy level (at a level above), does not implement so.

Recommendation(s): Implement a rate-limiting `Service`, add it to the `ProxyService`. See for example this.

Status: Acknowledged

Update from the client: Rollup Boost is not intended to be publicly exposed. Relays or other trusted components should write directly to the proxy service. Chain operators should ensure that rate limiting is enforced elsewhere in the stack.

7.7 [Low] Use of `sync::Cache` in an async context

File(s): `src/server.rs`

Description: `moka` provides both `sync::Cache` and `future::Cache`, for use in multi-threaded and asynchronous contexts, respectively. While slightly more (9%) performant, `sync::Cache` carries some risk of blocking the async executor threads and therefore, the whole rollup-boost operation. See a more detailed discussion here. As of currently, the employed `Cache` methods carry a very low risk of temporarily blocking the executor.

Recommendation(s): Replace `moka::sync::Cache` by `moka::future::Cache`. Performance analysis comparing both.

Status: Fixed

Update from the client: Fixed in 9955e90

7.8 [Info] CI GitHub Action actions-rs unmaintained

File(s): `CI files`

Description: `actions-rs/toolchain` is a GitHub Action that provides `cargo` features (build, test, lint, etc.). As of currently, it is unmaintained and archived, and as such, any possible vulnerabilities will not be fixed.

Recommendation(s): Use `cargo` as already provided by the CI image `ubuntu-latest`, or explore other maintained Rust/ `cargo` GitHub Actions.

Status: Fixed

Update from the client: Fixed in a0c7625

7.9 [Info] Code inside the `is_fallback_enabled()` branch may not be reachable

File(s): src/servers.rs

Description: The function `RollupBoostServer.get_payload` ensures the execution mode is enabled before attempting to call the builder. If the execution mode disables payload retrieval (e.g., dry-run mode), the builder is not called.

```
async fn get_payload(
    &self,
    payload_id: PayloadId,
    version: Version,
) -> RpcResult<OpExecutionPayloadEnvelope> {
    let l2_client_future = self.l2_client.get_payload(payload_id, version);
    let builder_client_future = Box::pin(async move {
    let execution_mode = self.execution_mode();
    // @audit @audit When fallback mode is enabled, the builder is skipped entirely
    if !execution_mode.is_get_payload_enabled() {
        info!(message = "dry run mode is enabled, skipping get payload builder call");

        // We are in dry run mode, so we do not want to call the builder.
        return Err(ErrorObject::owned(
            INVALID_REQUEST_CODE,
            "Dry run mode is enabled",
            None::<String>,
        ));
    }
    // ...
```

After, the function concurrently awaits both the L2 client and the (potential) builder results using `tokio::join!`. It then matches the outcomes:

- If both the builder and the L2 client succeed (`Ok` results), the function checks if fallback mode is enabled:;

- If fallback mode (`is_fallback_enabled()`) is enabled, the L2 client's payload is chosen. - Otherwise, the builder's payload is used.

- If only the L2 client succeeds while the builder fails or returns no payload, the L2 client's payload is selected instead;

```
    let (l2_payload, builder_payload) = tokio::join!(l2_client_future, builder_client_future);
    let (payload, context) = match (builder_payload, l2_payload) {
        (Ok(Some(builder)), Ok(l2_payload)) => {
            if self.execution_mode().is_fallback_enabled() {
                // @audit: This branch will never be executed
                // Default to op-geth's payload
                Ok((l2_payload, PayloadSource::L2))
            } else {
                Ok((builder, PayloadSource::Builder))
            }
        }
        (_, Ok(l2)) => Ok((l2, PayloadSource::L2)),
        (_, Err(e)) => Err(e),
    }?;
```

Let's say the `execution_mode` is changed to fallback (i.e., `self.execution_mode().is_fallback_enabled()` is `true`) before the `get_payload` function is invoked. Then, `builder_client_future` would contains an error meaning that the dry run mode is enabled.

The code inside the `is_fallback_enabled()` branch is unreachable because if fallback mode is active, the builder is never called in the first place — it is skipped earlier when checking `is_get_payload_enabled()`. Although this branch is not executed, it does not cause incorrect behavior. The fallback logic properly defaults to selecting the L2 payload when needed.

Recommendation(s): Review the intended flow to ensure the unused code executes when fallback execution mode is enabled or remove it to reduce its complexity.

Status: Fixed

Update from the client: Fixed in the PR 198

7.10 [Info] Incoming requests have a large size limit

File(s): `src/proxy.rs`

Description: The body of all incoming HTTP requests are first passed through the function `http_helpers::read_body()` which returns the encoded JSON bytes, if valid in the size range. Internally, this function copies the bytes into a new, returned, buffer. Given that the size limit is currently set at `u32::MAX`, much larger than expected requests, extra-large (and likely invalid) requests can be used to cause denial-of-service due to the resources needed to copy all of the incoming data. Moreover, the header size of incoming HTTP requests is not set.

Recommendation(s): Limit the size limit to values closer to expected requests. On the header size, set the maximum header size in order to prevent issues related with oversized headers.

Status: Fixed

Update from the Nethermind Security: The fixing for this issue was identified in PR 221.

7.11 [Info] `boost_sync` flag can cause discrepancies between builder and L2 op-geth

File(s): `src/server.rs`

Description: Here is the relevant part for handling FCU requests:

```
    // TODO: Use _is_block_building_call to log the correct message during the async call to builder
        let (should_send_to_builder, _is_block_building_call) =
            if let Some(attr) = payload_attributes.as_ref() {
                // payload attributes are present. It is a FCU call to start block building
                // Do not send to builder if no_tx_pool is set, meaning that the CL node wants
                // a deterministic block without txs. We let the fallback EL node compute those.
                let use_tx_pool = !attr.no_tx_pool.unwrap_or_default();
                (use_tx_pool, true)
            } else {
                // no payload attributes. It is a FCU call to lock the head block
                // previously synced with the new_payload_v3 call. Only send to builder if boost_sync is enabled
                (self.boost_sync, false)
            };

        let execution_mode = self.execution_mode();
        let trace_id = span.id();
        if let Some(payload_id) = l2_response.payload_id {
            self.payload_trace_context.store(
                payload_id,
                fork_choice_state.head_block_hash,
                payload_attributes.is_some(),
                trace_id,
            );
        }

        if execution_mode.is_disabled() {
            debug!(message = "execution mode is disabled, skipping FCU call to builder", "head_block_hash" =
            ↪ %fork_choice_state.head_block_hash);
        } else if should_send_to_builder {
            let builder_client = self.builder_client.clone();
            tokio::spawn(async move {
                let _ = builder_client
                    .fork_choice_updated_v3(fork_choice_state, payload_attributes.clone())
                    .await;
            });
        } else {
            info!(message = "no payload attributes provided or no_tx_pool is set", "head_block_hash" =
            ↪ %fork_choice_state.head_block_hash);
        }
```

Let's focus on the following scenario:

1. `execution_mode` is `ExecutionMode::Enabled`;
2. FCU is NOT a building block, but an update head;
3. If `self.boost_sync` is disabled, the builder won't receive the update head from the Sequencer; it will have to wait for it from its own L2 node, which can be slow(er);
4. The request for block building comes before the builder has had time to update the head, which results in the builder not building a block;

A similar argument can be made for the `NewPayload` call because it depends on `self.boost_sync`. This may be an even bigger issue because the builder could take too long to receive this new block via P2P (even though it's physically close to the Sequencer).

Recommendation(s): Maybe remove `self.boost_sync` altogether. There is no massive benefit in having this flag; conversely, it complicates logic and may lead to bugs.

If this is not acceptable, we recommend that if `ExecutionMode::Enabled` *overpowers* `self.boost_sync`, i.e., even if `boost_sync` *is* disabled, we forward requests for `ExecutionMode::Enabled`. This later is not the best solution because logic is still unnecessarily complex, and issues can arise (albeit highly unlikely) when `ExecutionMode` is changed and `boost_sync` is disabled.

Status: Fixed

Update from the client: Fixed in the PR 206

7.12 [Info] getPayload requests to builder when FCU `no_tx_pool` is true

File(s): `src/server.rs`

Description: FCUs with attributes have a `no_tx_pool` field. While handling FCUs, the ones with true `no_tx_pool` are not sent to the builder. The code is presented below:

```
    let (should_send_to_builder, _is_block_building_call) =
        if let Some(attr) = payload_attributes.as_ref() {
            // payload attributes are present. It is a FCU call to start block building
            // Do not send to builder if no_tx_pool is set, meaning that the CL node wants
            // a deterministic block without txs. We let the fallback EL node compute those.
            let use_tx_pool = !attr.no_tx_pool.unwrap_or_default();
            (use_tx_pool, true)
        } // ...
    }
    // ...
    else if should_send_to_builder {
        let builder_client = self.builder_client.clone();
        tokio::spawn(async move {
            let _ = builder_client
                .fork_choice_updated_v3(fork_choice_state, payload_attributes.clone())
                .await;
        });
    }
```

However, after this, in `get_payload` the rollup-boost will attempt to fetch a block from the builder even if `no_tx_pool` was true, resulting in an unknown payload error from the builder. The code in the `get_payload` function is presented below:

```
    if !self.payload_trace_context.has_attributes(&payload_id) {
        // block builder won't build a block without attributes
        info!(message = "no attributes found, skipping get_payload call to builder");
        return Ok(None);
    }

    let builder = self.builder_client.clone();
    let payload = builder.get_payload(payload_id, version).await?;
```

Recommendation(s): Do not try to fetch payloads from the builder associated with FCUs with attributes and with true `no_tx_pool`. Leverage/update the use of the trace context accordingly.

Status: Fixed

Update from the client: Fixed in PR 195

7.13 [Best Practice] Concurrency in server

File(s): src/server.rs

Description: Incoming RPC requests are handled by spawning new asynchronous tasks with `tokio::spawn`; the code doesn't show any mechanism to limit the number of concurrently executed tasks.

Recommendation(s): Implement limiting, e.g., Semaphore might be a solution.

Status: Acknowledged

Update from the client: There are instances where op-node will send a quick succession of FCUs following derivation which requires that no rate limiting is in place. Further, Tokio is setup by default with number of background workers equal to number of CPU cores and tokio tasks are efficient to spawn. Relays or other trusted components should write directly to the proxy service. Chain operators should ensure that rate limiting is enforced elsewhere in the stack. We believe this finding should not be in the report.

7.14 [Best Practice] Different conditions for cache insertion/eviction

File(s): `src/server.rs`

Description: While payload IDs are stored in the cache on every FCU call, they are only removed in new-payload calls when `boost_sync` is enabled and the current execution mode is not `Disabled`. If the second set of conditions is not fulfilled, then the cache is only evicted upon its size limit being reached (100 entries).

Recommendation(s): To avoid an unnecessary large cache, only keep track of payload IDs for FCUs which are sent to the builder.

Status: Fixed

Update from the client: Fixed in 07b7f70

7.15 [Best Practice] Do not clone payload in server|fcu_v3

File(s): server.rs

Description: The cloned code in the `server::fork_choice_updated_v3` function (link/src/server.rsL302)) is not needed:

```
let _ = builder_client
    .fork_choice_updated_v3(fork_choice_state, payload_attributes.clone())
    .await;
```

Recommendation(s): We suggest not cloning the `payload_attributes`:

```
- let _ = builder_client
-     .fork_choice_updated_v3(fork_choice_state, payload_attributes.clone())
-     .await;
+ let _ = builder_client
+     .fork_choice_updated_v3(fork_choice_state, payload_attributes)
+     .await;
```

Status: Fixed

Update from the client: Fixed in PR 210

7.16 [Best Practice] Silent failures of `tokio::spawn`

File(s): server.rs

Description: In `fork_choice_updated_v3` and `new_payload` functions, requests to the builder are dispatched asynchronously using `tokio::spawn` to ensure quick server responses. However, there are some concerns:

In the `new_payload` function, the result of the spawned task is explicitly ignored:

```
    tokio::spawn(async move {
        let _ = builder.new_payload(new_payload_clone).await;
    });
```

Recommendation(s): Failures in the spawned tasks should be logged using tracing or captured through metrics to aid in debugging and monitoring.

Status: Acknowledged

Update from the client: The underlying methods (`new_payload_v3` and `new_payload_v4`) are instrumented with `tracing` and they are dispatched early.

7.17 [Best Practice] Using expect and unwrap in the code

File(s): src/client/auth.rs src/client/http.rs

Description: There are unchecked unwrap and expect calls in few non-test paths which can trigger the process to panic! if it fails to unwrap.

Recommendation(s): Replace unwrap/expect with proper error handling

Status: Fixed

Update from the client: Fixed in PR 243. This PR resolved most unwraps and expects in production code. Some unwraps remain in test code. And some expects remain in prod code - but are limited to service initialization or cases where the panic case is guarded against.

7.18 [Best Practice] CI use cargo deny

File(s): .

Description: `cargo deny` is a dependency linter, which analyzes dependencies security advisories, bans, licenses, and sources. If added to CI, such analysis will occur when the respective pipeline is triggered, enabling an alert, if there are any issues with a newly added dependency, or if any new issues popped-up with existing dependencies. There is no apparent dependency scanning or vulnerability management visible in the repository.

Recommendation(s): Add `cargo deny` or any other dependency vulnerability scanner to a CI pipeline.

Status: Fixed

Update from the client: Fixed in PR 249

7.19 [Best Practice] Do not clone payload in rpc|new_payload_v3

File(s): `rpc.rs`

Description: In `rpc.rs|new_payload_v3`, the `payload` is unnecessarily cloned, just to log `hash`. The average block size on, e.g., Base (which is another op-stack derived chain), is 65KB, while max was 500KB (source), this could easily be avoided.

Recommendation(s): We recommend to apply the following change:

```
- let execution_payload = ExecutionPayload::from(payload.clone());
- let block_hash = execution_payload.block_hash();
+ // Note: this can be hidden behind a function call to improve the readeability
+ let block_hash = payload.payload_inner.payload_inner.block_hash;
```

Note: Similarly for cloning the `new_payload` here.

Status: Fixed

Update from the client: Fixed in PR 210

7.20 [Best Practice] Logic of forwarding requests should be improved

File(s): `proxy.rs`

Description: All requests which are to be forwarded should not be treated equally. Specifically, there are `eth_` and `miner_` requests. If for some reason `miner_` succeeds on the L2-EL but fails on the builder (or vice versa), it can result in discrepancy with critical information for consensus, such as `GasPrice` and `GasLimit`.

This can lead to a block created by the builder with a higher gas limit than what L2-EL expects. As a result, the L2-EL will mark this block invalid and potentially flag all the future builder blocks as invalid invalid as well, effectively disabling builder functionality.

Recommendation(s): Increase the robustness of the forwarding mechanism. The `miner_` should be forwarder to the builder only if the call to EL-L2 succeeds. If the builder fails to process the request, the failure should be logged. In addition, the system should set the `execution_status` to `Disabled` to prevent further issues.

Status: Acknowledged

Update from the client: During the audit, issue 7.20 was raised regarding how we forwarded requests in `proxy.rs`. At the time, all forwarded methods were handled the same way. Nethermind flagged that this could cause an issue if `miner_` methods like `miner_setGasLimit` succeeded on the L2 execution client but failed on the builder, it could lead to inconsistent blocks constraints.

We addressed this in PR 221 by updating the forwarding logic to include retries for `miner_setMaxDASize`. If the builder request fails, the updated logic set `execution_mode` to `Disabled` and continually retried the request. However, this PR introduced a memory leak. The retry logic for `miner_setMaxDASize` would clone the request and sleep 200ms between attempts. In cases where the call was timing out, this led to a large buildup of in-flight retries and unbounded memory growth.

We fixed this in PR 302 which removes the retry logic for `miner_setMaxDASize` entirely. Instead, these calls are now treated like any other forwarded request. This is sufficient because `op-batcher` sends a `miner_setMaxDASize` update every 10 seconds regardless of the current throttle state, so the builder receives updates regularly even if a single call fails, removing the need for retry logic in the proxy.

Since this fix landed, Flashblocks logic has been added to the codebase and the project has been restructured into a Cargo workspace. The logic previously in scope for the audit is now under `crates/rollup-boost`.

8 Security Implications of Using Permissioneless Builders

The following issues are only applicable to permissionless builders. They arise because rollup-boost does not check the blocks built by builders but relies solely on **op-geth** to validate these blocks. Furthermore, the system in which Rollup-Boost operates is controlled by Raft consensus, which is not Byzantine resistant.

8.1 MEV extraction attack like the one performed against Mev-Boost

File(s): server.rs

Note: A related attack is described in the analysis of the MEV-Boost Relay Attack.

Description: Under the current setup, an attacker does not even need to exploit the system in a sophisticated way to achieve similar outcomes. They can freely insert or reorder transactions as they wish, without resistance or validation barriers.

All transactions are immediately proxied to the block builders. Builders can observe the transactions and (re)order them in any way they want or insert their MEV-extracting transactions. This is not against the protocol and will pass the *op-geth* validation.

Recommendation(s): We recommend one of the suggestions below to solve it:

- Allow only permissioned builders;
- Encrypted mempools (TEE);
- Maybe not full-blown TEE, but we could only reveal transactions to the builder after they commit to inclusion and ordering of transactions. If they don't respect the commitment, then rollup-boost fallbacks to *op-geth* block;

8.2 Do not blindly default to builder block

File(s): server.rs

Description: The current design blindly defaults to builder-block. This can be problematic as observed in the HA-doc:

> Note that rollup-boost does not currently feature a block selection policy and will optimistically select the builder's block for validation. In the event of a bug in the builder, it is possible valid but undesirable blocks (e.g. empty blocks) are produced. Without a block selection policy, rollup-boost will prefer these blocks over the default execution client. Proper monitoring alerting can help mitigate this but further designs should be explored to introduce safeguards into rollup-boost directly rather than relying on the builder implementation being correct.

Possible scenarios:

- Proxy fails (for whatever reason) to forward some transaction;
- Or, in the future, if builder breaks bad and censors transactions;

In both cases the block produced by the builder will be valid, but _bad_.

Recommendation(s): Since we have both builder-block and the L2-EL block, we could at least pick one which has higher `gas_used` or tx count as the quick solution to some of the problems mentioned above.

8.3 Slow down (or even disable) *op-geth* with extremely large block from builder

File(s): `server.rs`

Description: A bad block builder can create a huge block, which `op-geth` will try to validate. If the block is too large, this can significantly slow down the `op-geth` execution or even cause it to crash altogether.

Here is the code where `op-geth` validates the header, as can be seen on this line, for op-stack chains, validation of the block's `GasLimit` is skipped.

What an attacker (block builder gone bad) can do is prepare this huge block beforehand. It does not even have to be valid, because the Merkle root is compared only after all execution is done (link) to stall the network or knock out the L2-op-geth altogether.

Recommendation(s): Do not trust the block builder unquestioningly. Since we have a block from `op-geth` and a builder, we can perform some quick checks, e.g., that the `GasLimit` produced by the builder is not 100% more, or something similar.

9 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. There are various types of software documentation. Some of the most common types include:

- User manual: A user manual is a document that provides information about how to use the application. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities;
- Code documentation: Code documentation is a document that provides details about the code. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface). It includes details about the methods, parameters, and responses that can be used to interact with the system;
- Testing documentation: Testing documentation is a document that provides information about how the code is tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the application. This type of documentation is critical in ensuring that the system is secure and free from vulnerabilities.

These types of documentation are essential for software development and maintenance. They help ensure that software is properly designed, implemented and tested, and provide a reference for developers who need to modify or maintain the code in the future.

Remarks about the Rollup Boost documentation

The documentation for the Rollup Boost was provided in the folder docs with a high-level description. Moreover, the Rollup Boost team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

9.1 The code for `get_payload` is not easy to reason about

Code as documentation is crucial for ensuring readability and reducing the effort for program comprehension, especially in complex systems, where implicit logic can easily lead to misunderstandings or errors. The `get_payload` function in `server.rs` exemplifies this: the control flow is hard to trace, which can lead to bugs in the future. By not making the intent and decision-making process explicit in the code, future contributors may misinterpret the expected behavior. Some examples are listed below:

- Inconsistencies in how `execution_mode` is handled (see issue);
- Code inside the `is_fallback_enabled()` branch may not be reachable in certain scenarios (see issue), potentially resulting in future defects.

Recommendation(s): Refactor the `get_payload` function so that there is a clear difference in flow when a block is expected from the builder vs. when it is not. A proposed refactor is attached below, which improves the code by:

a. Ensuring `execution_state` is read once, preventing inconsistencies as described above;
b. Making the control flow more explicit: a block is fetched from the builder only when `ExecutionState::Enabled`;
c. Small performance gain, as we do not have to create a future and call `join` unnecessarily.

The diff below illustrates a suggestion of refactoring:

```
src/server.rs | 81 ++++++++++++++++++++++++++++++-------------------------
 1 file changed, 39 insertions(+), 42 deletions(-)

diff --git a/src/server.rs b/src/server.rs
index 59d83cd..bfdc83b 100644
--- a/src/server.rs
+++ b/src/server.rs
@@ -117,6 +117,7 @@ impl ExecutionMode {
         matches!(self, ExecutionMode::Disabled)
     }

+    #[allow(unused)]
     fn is_fallback_enabled(&self) -> bool {
         matches!(self, ExecutionMode::Fallback)
     }
@@ -536,26 +537,48 @@ impl RollupBoostServer {
         }
         Ok(self.l2_client.new_payload(new_payload).await?)
     }
-
     async fn get_payload(
         &self,
         payload_id: PayloadId,
         version: Version,
     ) -> RpcResult<OpExecutionPayloadEnvelope> {
+        let (payload, context) = if self.execution_mode().is_get_payload_enabled() {
+            self.get_payload_from_builder(payload_id, version).await?
+        } else {
+            info!(message = "builder mode not enabled, skipping get payload builder call");
+            let payload = self.l2_client.get_payload(payload_id, version).await?;
+            (payload, PayloadSource::L2)
+        };
+
+        tracing::Span::current().record("payload_source", context.to_string());
+        counter!("rpc.blocks_created", "source" => context.to_string()).increment(1);
+
+        let inner_payload = ExecutionPayload::from(payload.clone());
+        let block_hash = inner_payload.block_hash();
+        let block_number = inner_payload.block_number();
+
+        info!(
+            message = "returning block",
+            "hash" = %block_hash,
+            "number" = %block_number,
+            %context,
+            %payload_id,
+        );
+        Ok(payload)
+    }
+
+    async fn get_payload_from_builder(
+        &self,
+        payload_id: PayloadId,
+        version: Version,
+    ) -> RpcResult<(OpExecutionPayloadEnvelope, PayloadSource)> {
         let l2_client_future = self.l2_client.get_payload(payload_id, version);
         let builder_client_future = Box::pin(async move {
@@ -563,7 +586,7 @@ impl RollupBoostServer {
             if !self.payload_trace_context.has_attributes(&payload_id) {
                 info!(message = "no attributes found, skipping get_payload call to builder");
-                return Ok(None);
+                return Ok::<Option<OpExecutionPayloadEnvelope>, ErrorObject>(None);
             }

             let builder = self.builder_client.clone();
@@ -582,38 +605,12 @@ impl RollupBoostServer {
```

```
         let (l2_payload, builder_payload) = tokio::join!(l2_client_future, builder_client_future);
         let (payload, context) = match (builder_payload, l2_payload) {
-            (Ok(Some(builder)), Ok(l2_payload)) => {
-                if self.execution_mode().is_fallback_enabled() {
-                    Ok((l2_payload, PayloadSource::L2))
-                } else {
-                    Ok((builder, PayloadSource::Builder))
-                }
-            }
+            (Ok(Some(builder)), _) => Ok((builder, PayloadSource::Builder)),
             (_, Ok(l2)) => Ok((l2, PayloadSource::L2)),
             (_, Err(e)) => Err(e),
         }?;

-        tracing::Span::current().record("payload_source", context.to_string());
-        counter!("rpc.blocks_created", "source" => context.to_string()).increment(1);
-
-        let inner_payload = ExecutionPayload::from(payload.clone());
-        let block_hash = inner_payload.block_hash();
-        let block_number = inner_payload.block_number();
-
-        info!(
-            message = "returning block",
-            "hash" = %block_hash,
-            "number" = %block_number,
-            %context,
-            %payload_id,
-        );
-        Ok(payload)
+        Ok((payload, context))
     }
 }
```

10 Test Suite Evaluation

10.1 Code Coverage Analysis

Table 3 presents the code coverage with overall percentages exceeding 79% in three categories: function, line, and region metrics. However, branch coverage stands out as notably lower, achieving only 63.64% overall. This discrepancy suggests that **while most functions and lines of code are executed during testing, many conditional branches are not fully explored**. This can leave important edge cases and alternative code paths untested, potentially hiding bugs that only manifest under specific conditions. **Improving branch coverage** should be prioritized to enhance the robustness and reliability of the test suite, where coverage is especially low.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
bin/main.rs	100.00% (2/2)	100.00% (7/7)	100.00% (2/2)	– (0/0)
cli.rs	17.65% (3/17)	60.91% (67/110)	42.20% (46/109)	50.00% (6/12)
client/auth.rs	100.00% (6/6)	100.00% (33/33)	100.00% (6/6)	– (0/0)
client/http.rs	100.00% (3/3)	100.00% (27/27)	87.50% (7/8)	– (0/0)
client/rpc.rs	92.00% (23/25)	92.73% (153/165)	88.89% (72/81)	50.00% (2/4)
debug_api.rs	100.00% (14/14)	100.00% (123/123)	90.16% (55/61)	– (0/0)
health.rs	100.00% (5/5)	100.00% (15/15)	100.00% (8/8)	100.00% (2/2)
metrics.rs	50.00% (3/6)	40.91% (18/44)	38.89% (14/36)	25.00% (1/4)
proxy.rs	94.34% (50/53)	96.32% (707/734)	86.76% (249/287)	62.50% (5/8)
server.rs	95.38% (62/65)	91.77% (591/644)	85.59% (285/333)	76.67% (23/30)
tracing.rs	71.43% (5/7)	72.50% (87/120)	65.12% (28/43)	50.00% (3/6)
Totals	**86.70% (176/203)**	**90.41% (1828/2022)**	**79.26% (772/974)**	**63.64% (42/66)**

Table 3: Test Coverage Summary

A. Final Remarks: Code coverage should be increased

The code has a good level of line coverage, but some branches are not executed in tests. There are several scenarios that should be tested in the application:

- `client/rpc.rs`: The client does not validate or handle the case where `new_payload_v4` returns an invalid status, leaving the `RpcClientError::InvalidPayload` path untested.
- `client/rpc.rs`: The success path for `new_payload_v4` is not exercised, which risks silently breaking correct handling of valid responses.
- `client/rpc.rs`: The handling of `get_payload_v4` receiving a V3 payload is untested; if this happens, the client should emit a version mismatch error.
- `proxy.rs`: The proxy does not have coverage for cases where the HTTP body is unreadable, e.g., truncated or disconnected payloads, which should trigger JSON-RPC error -32700.
- `proxy.rs`: The proxy does not validate or test scenarios where the HTTP body contains syntactically invalid JSON, which should result in a parse error (-32700).
- `proxy.rs`: Valid JSON-RPC requests with unknown method names are not tested and may return incorrect or missing errors instead of the required -32601 error.

Recommendation(s): Consider implementing tests for the scenarios presented above.

10.2 Tests Output

```
cargo test --verbose --features ""
 ...
running 13 tests
test client::rpc::tests::valid_jwt ... ok
test client::rpc::tests::invalid_jwt ... ok
test debug_api::tests::test_debug_client ... ok
test client::rpc::tests::test_invalid_args ... ok
test proxy::tests::test_proxy_service ... ok
test server::tests::test_server ... ok
test proxy::tests::test_forward_miner_set_extra ... ok
test proxy::tests::test_direct_forward_mock_request ... ok
test proxy::tests::test_forward_eth_send_raw_transaction ... ok
test proxy::tests::test_forward_eth_send_raw_transaction_conditional ... ok
test proxy::tests::test_forward_set_max_da_size ... ok
test proxy::tests::test_forward_miner_set_gas_limit ... ok
test proxy::tests::test_forward_miner_set_gas_price ... ok

test result: ok. 13 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 3.05s
     Running `worlrollup-boost/target/debug/deps/rollup_boost-fbfdc57792a0
     9884`
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
     Running `worlrollup-boost/target/debug/deps/builder_full_delay-bfc796
     ffe204c09c`
running 1 test
test builder_full_delay ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 18.05s
     Running `worlrollup-boost/target/debug/deps/builder_returns_incorrect_
     block-3405f3e0ee2936c7`

running 1 test
test builder_returns_incorrect_block ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 8.94s
     Running `worlrollup-boost/target/debug/deps/execution_mode-ca3f9149ef
     504a5c`

running 1 test
test execution_mode ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 18.76s
     Running `worlrollup-boost/target/debug/deps/no_tx_pool-4d8b833e9d988b
     38`

running 1 test
test no_tx_pool ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 10.78s
     Running `worlrollup-boost/target/debug/deps/remote_builder_down-a30e4c
     733d173d77`

running 1 test
test remote_builder_down ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 21.87s
     Running `worlrollup-boost/target/debug/deps/simple-7feb4dac992462ca`

running 1 test
test test_integration_simple ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 10.73s
```

11 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.