Security Review Report
NM-0411-0509 WORLD-CHAIN-TX-PROXY

NETHERMIND
SECURITY

(May 27, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for World Chain. The audit review focused on Transaction Proxy Relayer, which is a simple pass-through proxy that accepts RPC requests and redirects the traffic to 3 builder back-ends that perform a custom validation.

The audit comprises 879 lines of rust code. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

Along this document, we report 14 points of attention, where they are classified as one `High`, four `Medium`, one `Low`, six `Informational`, and two `Best Practices`. The issues are summarized in Fig. 1.

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

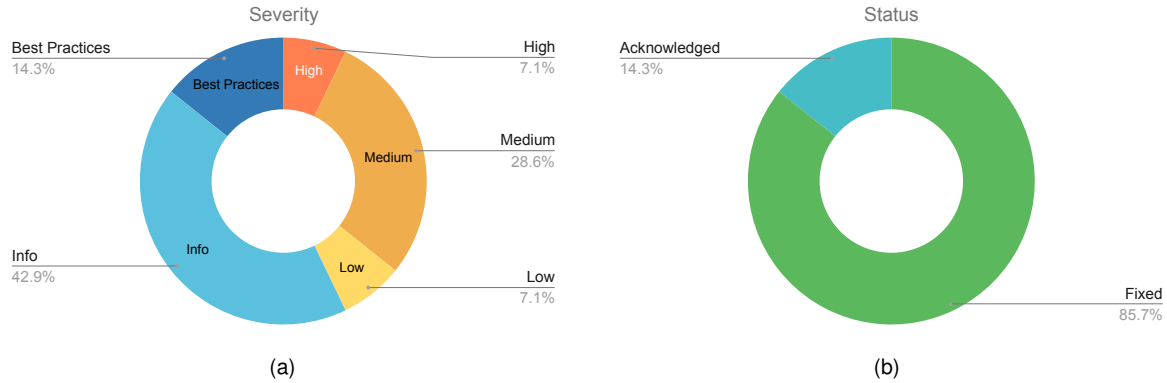

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (4), **Low** (1), **Undetermined** (0), **Informational** (6), **Best Practices** (2). **Distribution of status: Fixed** (12), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 1, 2025
Response from Client	Regular responses during audit engagement
Final Report	May 27, 2025
Repositories	tx-proxy
Initial Commit	0fac2d2
Final Commit	9cdbe54
Documentation	Readme file
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	lib.rs	6	1	16.7%	0	7
2	fanout.rs	24	5	20.8%	4	33
3	rpc.rs	99	5	5.1%	15	119
4	proxy.rs	360	16	4.4%	60	436
5	cli.rs	252	42	16.7%	49	343
6	client.rs	54	2	3.7%	9	65
7	validation.rs	76	7	9.2%	15	98
8	bin/main.rs	8	1	12.5%	1	10
	Total	**879**	**79**	**9.0%**	**153**	**1111**

3 Summary of Issues

	Finding	Severity	Update
1	Unbounded request size allows resource exhaustion and denial of service	High	Fixed
2	Absence of rate limiting allows malicious users to monopolize the maximum concurrent connections	Medium	Acknowledged
3	Clients are allowed to use HTTP, exposing JWT tokens	Medium	Acknowledged
4	HTTP clients are instantiated without applying timeout	Medium	Fixed
5	The graceful shutdown is not handled completely.	Medium	Fixed
6	Metrics server could fail to start, but won't stop the server	Low	Fixed
7	Address parsing in metrics server fails for IPv6 addresses	Info	Fixed
8	Flags cannot be parsed from `.env` file	Info	Fixed
9	Missing `[instrument]` in `ValidationService`	Info	Fixed
10	Mixing `HttpBody` with `Hyper` may cause future compatibility issues	Info	Fixed
11	`max_concurrent_connections` cannot be configured via environment variables	Info	Fixed
12	cargo-audit: crossbeam-channel: double free on Drop	Info	Fixed
13	Tracing should be configured to write logs to a file	Best Practices	Fixed
14	Unnecessary cloning	Best Practices	Fixed

4 System Overview

`TX-Proxy` is a simple pass-through proxy that accepts RPC requests of two types (`eth_sendRawTransaction`, `eth_sendRawTransactionConditional`) and redirects the traffic to 3 builder back-ends. This is needed since the builder back-end provides a custom validation policy for PBH transactions. If validation was successful on all builders, the request is redirected to Alchemy's L2 relays. The TX proxy relayer is behind Alchemy's proxy.

4.1 Overview of the codebase

The relayer uses the `Tower` architecture, which allows modular and composable services. There are two back-ends that each request passes through, and each back-end spawns three concurrent HTTP clients.

- **HealthLayer (Out of Scope)**: A simple middleware that filters the `/healthz` endpoint.
- **Validation Layer**: Receives incoming requests (after being forwarded by the HealthLayer) and fans them out to the Builder back-ends. The Builder backends (out of scope) apply PBH validation and return a success flag. If validation is successful, the request is forwarded to the ProxyLayer.
- **Proxy Layer**: A minimal layer that fans out the request to L2 nodes and returns the first response.

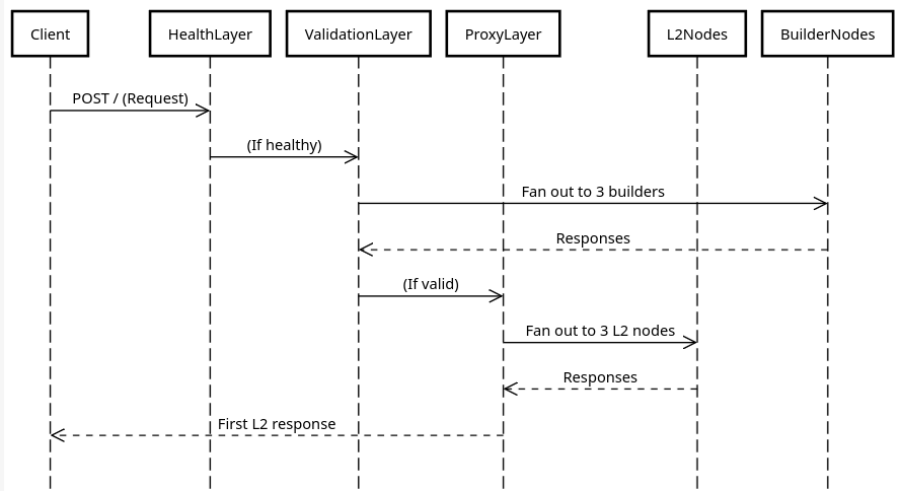

Figure 1: Request flow

5 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.

6 Issues

6.1 [High] Unbounded request size allows resource exhaustion and denial of service

File(s): rpc.rs

Description: Upon receiving an HTTP request, the `from_request` is invoked to parse and read the body:

```
pub async fn from_request(request: http::Request<HttpBody>) -> Result<Self> {
    let (parts, body) = request.into_parts();
    let (body_bytes, _) = http_helpers::read_body(&parts.headers, body, u32::MAX).await?;
    let method = serde_json::from_slice::<Request>(&body_bytes)?
        .method
        .to_string();

    Ok(Self {
        parts,
        body: body_bytes,
        method,
    })
}
```

The current implementation accepts payloads up to `u32::MAX` (approximately 4 GB), which is unnecessarily large for typical RPC requests. This exposes the server to a denial-of-service (DoS) attack, where a malicious user can send repeated large requests to consume memory and processing resources, potentially degrading service availability.

Recommendation(s): RPC requests are typically lightweight and rarely exceed a few megabytes. It is recommended to set a much lower and reasonable upper limit on the request body size

Status: Fixed.

Update from the client: Fixed in commit 01dadab

6.2 [Medium] Absence of rate limiting allows malicious users to monopolize the maximum concurrent connections

File(s): cli.rs

Description: While the server is configured to enforce a maximum number of concurrent connections, no rate limiting mechanism is implemented to regulate incoming traffic:

```
pub async fn run(self) -> Result<()> {
    // --SNIP
    let server = Server::builder()
        .set_http_middleware(middleware)
        .max_connections(self.max_concurrent_connections)
        .build(format!("{}:{}", self.http_addr, self.http_port))
        .await?;
}
```

The rate limit is a strategy for limiting network traffic. It puts a cap on how often an IP address can repeat an action within a certain timeframe. The lack of rate limiting allows a single IP address to continuously send requests without restriction, exhausting all available concurrent connections. This can lead to a denial of service (DoS) for legitimate users.

Status: Acknowledged.

Update from the client: The service is intended to run behind IP whitelisting and is only accessible within permissioned infrastructure. Since it is expected to receive high request volumes from a small set of IP addresses, rate limiting should be handled elsewhere in the stack.

6.3 [Medium] Clients are allowed to use HTTP, exposing JWT tokens

File(s): client.rs

Description: When initializing HTTP clients for the builder and L2 backends, the configuration permits both HTTPS and HTTP traffic:

```
impl HttpClient {
    pub fn new(url: Uri, secret: JwtSecret) -> Self {
        let connector = hyper_rustls::HttpsConnectorBuilder::new()
            .with_native_roots()
            .expect("no native root CA certificates found")
==>            .https_or_http()
            .enable_http1()
            .enable_http2()
            .build();
    }
}
```

On each request, a JWT token is attached for authentication:

```
let client = Client::builder(TokioExecutor::new()).build(connector);

let client = ServiceBuilder::new()
    .layer(DecompressionLayer::new())
    .layer(AuthClientLayer::new(secret))
    .service(client);

Self { client, url }
```

Allowing HTTP traffic poses a significant security risk, as HTTP does not encrypt requests. In non-local deployments—where the builder and L2 backends are hosted on separate servers—an attacker could intercept HTTP traffic and extract the JWT token, compromising the authentication mechanism.

Recommendation(s): The recommended approach is to explicitly disable HTTP support by requiring HTTPS connections only. If HTTP support is necessary for specific environments, consider encrypting the JWT token on the request.

Status: Acknowledged.

Update from the client: The service is intended to run within a permissioned infrastructure where all network traffic is secured via TLS. While HTTP is supported to enable local testing, production deployments are expected to use HTTPS exclusively. JWTs are only transmitted over these secure connections.

6.4 [Medium] HTTP clients are instantiated without applying timeout

File(s): cli.rs

Description: For each backend, three instances of an HTTP client are created. Each instance is expected to enforce a timeout value, which is configurable via a command-line flag during backend initialization to prevent potential resource exhaustion:

```
macro_rules! define_rpc_args {
    ($(($name:ident, $prefix:ident)),*) => {
        // --SNIP
        #[arg(long, env, default_value_t = 1000)]
        pub [<$prefix _timeout>]: u64,
    }
}
```

However, the configured timeout is not applied when initializing the HTTP clients:

```
pub fn build(&self) -> Result<FanoutWrite> {
    let jwt = self.get_jwt()?;
    let client_0 = HttpClient::new(self.[<$prefix _url_0>].clone(), jwt.clone());
    let client_1 = HttpClient::new(self.[<$prefix _url_1>].clone(), jwt.clone());
    let client_2 = HttpClient::new(self.[<$prefix _url_2>].clone(), jwt);
    Ok(FanoutWrite::new(vec![client_0, client_1, client_2]))
}
```

This omission could lead to unbounded request times, risking performance degradation or backend overload.

Recommendation(s): Consider applying the timeout when initializing the HTTP clients.

Status: Fixed.

Update from the client: Fixed in commit 19597ab

6.5 [Medium] The graceful shutdown is not handled completely.

File(s): cli.rs

Description: After the relayer server is started, a Tokio `select!` block is used to listen for scenarios where the server should shut down gracefully by invoking the handle's `stop()` method:

```
pub async fn run(self) -> Result<()> {
    // --SNIP
    let handle = server.start(RpcModule::new(()));

    let stopped_handle = handle.clone();
    let shutdown_handle = handle.clone();

    tokio::select! {
        _ = stopped_handle.stopped() => {
            error!("Server stopped unexpectedly or crashed");
            Err(eyre::eyre!("Server stopped unexpectedly or crashed"))
    }
        _ = tokio::signal::ctrl_c() => {
            error!("Received Ctrl-C, shutting down...");
            shutdown_handle.stop()?;
            Ok(())
        }
    }
}
```

The current implementation does not handle the `SIGTERM` signal, which is commonly used by process managers and container orchestrators in production environments to trigger a graceful shutdown. Since `SIGTERM` is not captured, the server may be terminated abruptly, skipping the invocation of `handle.stop()` and potentially resulting in unclean resource cleanup or data corruption.

Recommendation(s): Consider adding the `SIGTERM` arm in the `select!` block:

```
pub async fn run(self) -> Result<()> {
    // --SNIP
    let handle = server.start(RpcModule::new(()));

+   let mut sigterm = signal(SignalKind::terminate()).unwrap();

    let stopped_handle = handle.clone();
    let shutdown_handle = handle.clone();
+   let force_handle = handle.clone();

    tokio::select! {
        _ = stopped_handle.stopped() => {
            error!("Server stopped unexpectedly or crashed");
            Err(eyre::eyre!("Server stopped unexpectedly or crashed"))
    }
        _ = tokio::signal::ctrl_c() => {
            error!("Received Ctrl-C, shutting down...");
            shutdown_handle.stop()?;
            Ok(())
        }
+       _ = sigterm.recv() => {
+           error!("Forcefully shutdown by process manager");
+           force_handle.stop()?;
+           Ok(())
+       }
    }
}
```

Status: Fixed.

Update from the client: Fixed in commit 029d8dc

6.6 [Low] Metrics server could fail to start, but won't stop the server

File(s): cli.rs

Description: During server initialization, the Prometheus metrics server is launched asynchronously in a separate tokio task. However, this task's execution status is not monitored or validated. Consequently, any failures occurring during initialization of the metrics server would remain undetected, more specifically if the address passed is already bound to a TCP listener, potentially causing degraded observability or unexpected behavior. Furthermore, such initialization failures do not prevent the main server from continuing execution, potentially masking critical operational errors.

1) We initialize our metrics in a separate tokio task in `cli::Cli::init_metrics`

```
    fn init_metrics(&self) -> Result<()> {
        if self.metrics {
            ...
            tokio::spawn(init_metrics_server(addr, handle)); // Run the metrics server in a separate task
        }

        Ok(())
    }
```

2) `init_metrics_server` function starts the server, and in the case of errors, returns it as a `eyre::Result<()>`, which is not handled properly after

```
pub(crate) async fn init_metrics_server(
    addr: SocketAddr,
    handle: PrometheusHandle,
) -> eyre::Result<()> {
    let listener = TcpListener::bind(addr).await?;
    info!("Metrics server running on {}", addr);

    loop {
        match listener.accept().await {
            Ok((stream, _)) => {
                let handle = handle.clone(); // Clone the handle for each connection
                tokio::task::spawn(async move {
                    let service = service_fn(move |_req: Request<hyper::body::Incoming>| {
                        let response = match _req.uri().path() {
                            "/metrics" => Response::builder()
                                .header("content-type", "text/plain")
                                .body(HttpBody::from(handle.render()))
                                .unwrap(),
                            _ => Response::builder()
                                .status(StatusCode::NOT_FOUND)
                                .body(HttpBody::empty())
                                .unwrap(),
                        };
                        async { Ok::<_, hyper::Error>(response) }
                    });

                    let io = TokioIo::new(stream);

                    if let Err(err) = http1::Builder::new().serve_connection(io, service).await {
                        error!(message = "Error serving metrics connection", error = %err);
                    }
                });
            }
            Err(e) => {
                error!(message = "Error accepting connection", error = %e);
            }
        }
    }
}
```

Recommendation(s): Consider reworking the metrics server initialization such that its failure is observed and handled accordingly. Instead of spawning `init_metrics_server` as an unbound task, you could wait on its return in a supervised task or use a special error channel (e.g., a `oneshot::Sender`) to return initialization failures to the main thread. That would allow the application to fail fast or log critical startup failures before it carries on. Alternatively, use a task supervisor pattern (e.g., `tokio::task::JoinHandle` with explicit `.await` and error propagation) or structured concurrency utilities like `tokio::select!` with cancellation guards to monitor and manage task-level failures such that observability is not silently lost.

Status: Fixed.

Update from the client: Fixed in commit 5ba91e2

6.7 [Info] Address parsing in metrics server fails for IPv6 addresses

File(s): cli.rs

Description: During metrics server initialization, the address is constructed using string formatting with `::`

```
        let server = Server::builder()
            .set_http_middleware(middleware)
            .max_connections(self.max_concurrent_connections)
            .build(format!("{}:{}", self.http_addr, self.http_port))
            .await?;
```

```
fn init_metrics(&self) -> Result<()> {
    if self.metrics {
        let recorder = PrometheusBuilder::new().build_recorder();
        let handle = recorder.handle();

        Stack::new(recorder)
            .push(PrefixLayer::new("tx-proxy"))
            .install()?;

        // Start the metrics server
==>        let metrics_addr = format!("{}:{}", self.metrics_host, self.metrics_port);
        let addr: SocketAddr = metrics_addr.parse()?;
        tokio::spawn(init_metrics_server(addr, handle)); // Run the metrics server in a separate task
    }

    Ok(())
}
```

This approach fails for IPv6 addresses, as the resulting string (e.g., `::1:9090`) is not a valid representation and will cause parsing errors.

Recommendation(s): `SocketAddr` already exposes a `new` function to build an address from the ip and the port directly, consider using it.

Status: Fixed.

Update from the client: Fixed in commit e560eea

6.8 [Info] Flags cannot be parsed from `.env` file

File(s): cli.rs

Description: The server's configuration flags are expected to be loaded from environment variables. Based on the project structure and the presence of a `.env` file, it is also expected that these flags can be loaded automatically from the file. However, the `.env` file is not explicitly loaded in the main function.

While clap's `[arg(env)]` attribute can pick up environment variables already set in the shell session, it does not load a `.env` file by itself. As a result, variables defined only in `.env` will not be available when parsing the CLI arguments.

Recommendation(s): Consider using the `dotenv` crate and calling it at the very beginning of the main function

Status: Fixed.

Update from the client: Fixed in commit e5f8b8a

6.9 [Info] Missing `instrument` in `ValidationService`

File(s): validation.rs

Description: Unlike `forward` function in `HttpClient`, the `call` function in `ValidationService` manually injects `target = tx-proxy::validation` into each `debug!` statement. This approach is more verbose and prone to inconsistencies. Applying `[instrument]` would automatically apply span for all following traces.

Recommendation(s): Consider adding the `[instrument]` macro to the `call` function, similarly to how it is used in `forward`.

Example of a possible solution:

```
    #[instrument(
        skip(self, request),
        target = "tx-proxy::validation",
    )]
    fn call(&mut self, request: HttpRequest<HttpBody>) -> Self::Future {
```

Status: Fixed.

Update from the client: Fixed in commit c5c916c

6.10 [Info] Mixing `HttpBody` with `Hyper` may cause future compatibility issues

File(s): `cli.rs`

Description: The type `jsonrpsee::http_client::HttpBody` is used as the body type in the metrics HTTP handler, even though the handler itself is built using the Hyper crate. While this currently works, it creates a fragile dependency: if the internal implementation of `HttpBody` changes in future versions of `jsonrpsee`, the metrics server could break unexpectedly.

Recommendation(s): Consider using `hyper::Body directly` for HTTP handlers built with Hyper.

Status: Fixed.

Update from the client: Fixed in commit 5fa5ef2

6.11 [Info] `max_concurrent_connections` cannot be configured via environment variables

File(s): `cli.rs`

Description: The project uses the `clap` crate to fetch configuration values from either CLI arguments or environment variables. Although `MAX_CONCURRENT_CONNECTIONS` is defined in the `.env` file, the server currently does not accept its value from the environment due to the missing `env` attribute in the `clap` macro.

Recommendation(s): Consider adding `env` attribute to `clap` macro:

```
    #[clap(long = "http.max-concurrent-connections", default_value_t = 500)
+   ,env
    ]
    pub max_concurrent_connections: u32,
```

Status: Fixed.

Update from the client: Fixed in commit d304993

6.12 [Info] cargo-audit: crossbeam-channel: double free on Drop

File(s): `Cargo.toml`

Description: The project indirectly depends on `crossbeam-channel` version `0.5.14`, which is affected by a __double free on Drop__-vulnerability. This vulnerability can cause memory corruption and undefined behaviour. It was disclosed on 2025-04-08 and tracked under RUSTSEC-2025-0024.

To reproduce this issue, install cargo-audit locally and run `cargo audit` in the project's directory.

The dependency path is: `crossbeam-channel 0.5.14` -> `moka 0.12.10` -> `rollup-boost 0.1.0` -> `tx-proxy 0.1.0`

Recommendation(s): Consider updating dependencies to the versions that mitigate this issue. Also, as a possible solution, consider updating `Cargo.lock` by removing it and running `cargo build` once again.

Status: Fixed.

Update from the client: Fixed in commit ccd84da

6.13 [Best Practices] Tracing should be configured to write logs to a file

File(s): cli.rs

Description: The proxy relayer server is configured to trace incoming requests upon startup. However, the tracer is not configured with a custom writer, and thus defaults to writing logs to `stdout`:

```
impl<S> Default for Layer<S> {
    fn default() -> Self {
        // only enable ANSI when the feature is enabled, and the NO_COLOR
        // environment variable is unset or empty.
        let ansi = cfg!(feature = "ansi") && env::var("NO_COLOR").map_or(true, |v| v.is_empty());

        Layer {
            fmt_fields: format::DefaultFields::default(),
            fmt_event: format::Format::default(),
            fmt_span: format::FmtSpanConfig::default(),
            make_writer: io::stdout,
            is_ansi: ansi,
            log_internal_errors: false,
            _inner: PhantomData,
        }
    }
}
```

Writing logs solely to the terminal (`stdout`) means that all tracing information is lost if the server crashes or is restarted. Persisting logs to a file ensures that important operational and diagnostic information is retained across restarts.

Recommendation(s): Consider configuring a writer (File interface) to the tracing subscriber

Status: Fixed.

Update from the client: Fixed in commits 29b014a and 84c3455

6.14 [Best Practices] Unnecessary cloning

File(s): cli.rs

Description: In the `build` function of `define_rpc_args` macro implementation, the `jwt` token is cloned multiple times when constructing `HttpClient` instances:

```
pub fn build(&self) -> Result<FanoutWrite> {
    let jwt = self.get_jwt()?;
    let client_0 = HttpClient::new(self.[<$prefix _url_0>].clone(), jwt.clone());
    let client_1 = HttpClient::new(self.[<$prefix _url_1>].clone(), jwt.clone());
    let client_2 = HttpClient::new(self.[<$prefix _url_2>].clone(), jwt);
    Ok(FanoutWrite::new(vec![client_0, client_1, client_2]))
}
```

However, cloning `jwt` is unnecessary, since `JwtToken` implements the `Copy` trait, so we can simply pass it as a value.

Recommendation(s): Consider removing `.clone()`:

```
pub fn build(&self) -> Result<FanoutWrite> {
    let jwt = self.get_jwt()?;
    let client_0 = HttpClient::new(self.[<$prefix _url_0>].clone(), jwt);
    let client_1 = HttpClient::new(self.[<$prefix _url_1>].clone(), jwt);
    let client_2 = HttpClient::new(self.[<$prefix _url_2>].clone(), jwt);
    Ok(FanoutWrite::new(vec![client_0, client_1, client_2]))
}
```

Status: Fixed.

Update from the client: Fixed in commit f13af46

7 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. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the project's design and technical details. It includes information about the purpose of the project, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the project. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the project. 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) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was 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 smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

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

Remarks about WorldChain documentation

The `WorldChain` team has provided a comprehensive walkthrough of the project in the kick-off call. The Readme file, and the code natspec include the explanation of the intended functionalities. Moreover, the team addressed the questions and concerns raised by the Nethermind Security team, providing valuable insight and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
tx-proxy git:(0fac2d2)  cargo build --release
 Compiling proc-macro2 v1.0.94
 Compiling unicode-ident v1.0.18
 Compiling libc v0.2.171
 Compiling serde v1.0.219
 Compiling cfg-if v1.0.0
 Compiling autocfg v1.4.0
 Compiling itoa v1.0.15
 Compiling pin-project-lite v0.2.16
 Compiling memchr v2.7.4
 Compiling once_cell v1.21.3
 Compiling futures-core v0.3.31
 Compiling zerocopy v0.8.24
 Compiling shlex v1.3.0
 Compiling futures-sink v0.3.31
 Compiling log v0.4.27
 Compiling parking_lot_core v0.9.10
 Compiling scopeguard v1.2.0
 Compiling foldhash v0.1.5
 Compiling futures-io v0.3.31
 Compiling tracing-core v0.1.33
 Compiling futures-channel v0.3.31
 Compiling fnv v1.0.7
 Compiling futures-task v0.3.31
 Compiling lock_api v0.4.12
 Compiling slab v0.4.9
 Compiling pin-utils v0.1.0
 Compiling ryu v1.0.20
 Compiling version_check v0.9.5
 Compiling serde_json v1.0.140
 Compiling tower-service v0.3.3
 Compiling stable_deref_trait v1.2.0
 Compiling typenum v1.18.0
 Compiling thiserror v2.0.12
 Compiling httparse v1.10.1
 Compiling crunchy v0.2.3
 Compiling equivalent v1.0.2
 Compiling generic-array v0.14.7
 Compiling paste v1.0.15
 Compiling tiny-keccak v2.0.2
 Compiling unicode-xid v0.2.6
 Compiling getrandom v0.2.15
 Compiling jobserver v0.1.32
 Compiling rand_core v0.6.4
 Compiling cc v1.2.17
 Compiling mio v1.0.3
 Compiling signal-hook-registry v1.4.2
 Compiling quote v1.0.40
 Compiling socket2 v0.5.9
 Compiling syn v2.0.100
 Compiling cmake v0.1.54
 Compiling ruint-macro v1.2.1
 Compiling atomic-waker v1.1.2
 Compiling try-lock v0.2.5
 Compiling base64 v0.22.1
 Compiling fs_extra v1.3.0
 Compiling tower-layer v0.3.3
 Compiling dunce v1.0.5
 Compiling want v0.3.1
 Compiling icu_locid_transform_data v1.5.1
 Compiling writeable v0.5.5
 Compiling litemap v0.7.5
 Compiling httpdate v1.0.3
 Compiling percent-encoding v2.3.1
 Compiling ring v0.17.14
 Compiling aws-lc-sys v0.27.1
```

```
Compiling untrusted v0.9.0
Compiling icu_properties_data v1.5.1
Compiling num-traits v0.2.19
Compiling rustversion v1.0.20
Compiling aws-lc-rs v1.12.6
Compiling icu_normalizer_data v1.5.1
Compiling cpufeatures v0.2.17
Compiling indexmap v1.9.3
Compiling ident_case v1.0.1
Compiling strsim v0.11.1
Compiling bitflags v2.9.0
Compiling pkg-config v0.3.32
Compiling zeroize v1.8.1
Compiling ppv-lite86 v0.2.21
Compiling block-buffer v0.10.4
Compiling crypto-common v0.1.6
Compiling write16 v1.0.0
Compiling hex v0.4.3
Compiling utf8_iter v1.0.4
Compiling digest v0.10.7
Compiling rustls-pki-types v1.11.0
Compiling rand_chacha v0.3.1
Compiling utf16_iter v1.0.5
Compiling thiserror v1.0.69
Compiling core-foundation-sys v0.8.7
Compiling zstd-sys v2.0.15+zstd.1.5.7
Compiling sync_wrapper v1.0.2
Compiling hashbrown v0.12.3
Compiling crossbeam-utils v0.8.21
Compiling heck v0.5.0
Compiling rustls v0.23.25
Compiling core-foundation v0.10.0
Compiling security-framework-sys v2.14.0
Compiling form_urlencoded v1.2.1
Compiling mime v0.3.17
Compiling anyhow v1.0.97
Compiling crc-catalog v2.4.0
Compiling subtle v2.6.1
Compiling security-framework v3.2.0
Compiling crc v3.2.1
Compiling either v1.15.0
Compiling powerfmt v0.2.0
Compiling zstd-safe v7.2.4
Compiling num-conv v0.1.0
Compiling time-core v0.1.4
Compiling alloc-no-stdlib v2.0.4
Compiling itertools v0.13.0
Compiling time-macros v0.2.22
Compiling alloc-stdlib v0.2.2
Compiling deranged v0.4.1
Compiling num-integer v0.1.46
Compiling sha2 v0.10.8
Compiling ahash v0.8.11
Compiling adler2 v2.0.0
Compiling hashbrown v0.15.2
Compiling miniz_oxide v0.8.5
Compiling num-bigint v0.4.6
Compiling time v0.3.41
Compiling rustls-native-certs v0.8.1
Compiling indexmap v2.8.0
Compiling brotli-decompressor v4.0.2
Compiling synstructure v0.13.1
Compiling darling_core v0.20.11
Compiling crc32fast v1.4.2
Compiling toml_datetime v0.6.8
Compiling regex-syntax v0.6.29
Compiling matchit v0.7.3
Compiling winnow v0.7.4
Compiling ipnet v2.11.0
Compiling serde_derive v1.0.219
Compiling tokio-macros v2.5.0
Compiling tracing-attributes v0.1.28
Compiling futures-macro v0.3.31
```

```
Compiling zerofrom-derive v0.1.6
Compiling yoke-derive v0.7.5
Compiling tracing v0.1.41
Compiling futures-util v0.3.31
Compiling thiserror-impl v2.0.12
Compiling zerofrom v0.1.6
Compiling zerovec-derive v0.10.3
Compiling yoke v0.7.5
Compiling displaydoc v0.2.5
Compiling alloy-rlp-derive v0.3.11
Compiling derive_more-impl v2.0.1
Compiling zerovec v0.10.4
Compiling async-trait v0.1.88
Compiling icu_provider_macros v1.5.0
Compiling pin-project-internal v1.1.10
Compiling thiserror-impl v1.0.69
Compiling darling_macro v0.20.11
Compiling darling v0.20.11
Compiling tinystr v0.7.6
Compiling icu_locid v1.5.0
Compiling icu_collections v1.5.0
Compiling futures-executor v0.3.31
Compiling pin-project v1.1.10
Compiling derive_more-impl v1.0.0
Compiling ethereum_ssz_derive v0.8.3
Compiling opentelemetry v0.28.0
Compiling auto_impl v1.2.1
Compiling bytes v1.10.1
Compiling smallvec v1.14.0
Compiling rand v0.8.5
Compiling parking_lot v0.12.3
Compiling http v1.3.1
Compiling tokio v1.44.1
Compiling arrayvec v0.7.6
Compiling http-body v1.0.1
Compiling alloy-rlp v0.3.11
Compiling http-body-util v0.1.3
Compiling const-hex v1.14.0
Compiling derive_more v2.0.1
Compiling ruint v1.14.0
Compiling icu_provider v1.5.0
Compiling nybbles v0.3.4
Compiling derive_more v1.0.0
Compiling axum-core v0.4.5
Compiling jsonrpsee-types v0.24.9
Compiling prost-derive v0.13.5
Compiling alloy-primitives v0.8.25
Compiling icu_locid_transform v1.5.0
Compiling async-stream-impl v0.3.6
Compiling glob v0.3.2
Compiling utf8parse v0.2.2
Compiling rustc-hash v2.1.1
Compiling icu_properties v1.5.1
Compiling zerocopy v0.7.35
Compiling regex-syntax v0.8.5
Compiling getrandom v0.3.2
Compiling prost v0.13.5
Compiling ethereum_serde_utils v0.7.0
Compiling alloy-serde v0.12.6
Compiling ethereum_ssz v0.8.3
Compiling alloy-eip7702 v0.5.1
Compiling alloy-eip2930 v0.1.0
Compiling alloy-eip2124 v0.1.0
Compiling alloy-trie v0.7.9
Compiling anstyle-parse v0.2.6
Compiling alloy-eips v0.12.6
Compiling icu_normalizer v1.5.0
Compiling regex-automata v0.4.9
Compiling async-stream v0.3.6
Compiling nibble_vec v0.1.0
Compiling serde_urlencoded v0.7.1
Compiling futures v0.3.31
Compiling idna_adapter v1.2.0
```

```
Compiling simple_asn1 v0.6.3
Compiling idna v1.0.3
Compiling toml_edit v0.22.24
Compiling tokio-util v0.7.14
Compiling h2 v0.4.8
Compiling tokio-stream v0.1.17
Compiling tower v0.4.13
Compiling tower v0.5.2
Compiling url v2.5.4
Compiling axum v0.7.9
Compiling opentelemetry_sdk v0.28.0
Compiling alloy-consensus v0.12.6
Compiling jsonrpsee-core v0.24.9
Compiling regex-automata v0.1.10
Compiling strum_macros v0.27.1
Compiling hyper v1.6.0
Compiling brotli v7.0.0
Compiling flate2 v1.1.0
Compiling crossbeam-epoch v0.9.18
Compiling hyper-util v0.1.11
Compiling zstd v0.13.3
Compiling sha1 v0.10.6
Compiling pem v3.0.5
Compiling portable-atomic v1.11.0
Compiling anstyle-query v1.1.2
Compiling snap v1.1.1
Compiling overload v0.1.1
Compiling is_terminal_polyfill v1.70.1
Compiling endian-type v0.1.2
Compiling lazy_static v1.5.0
Compiling anstyle v1.0.10
Compiling colorchoice v1.0.3
Compiling radix_trie v0.2.1
Compiling anstream v0.6.18
Compiling sharded-slab v0.1.7
Compiling nu-ansi-term v0.46.0
Compiling jsonwebtoken v9.3.1
Compiling soketto v0.8.1
Compiling strum v0.27.1
Compiling matchers v0.1.0
Compiling hyper-timeout v0.5.2
Compiling reqwest v0.12.15
Compiling proc-macro-crate v3.3.0
Compiling tonic v0.12.3
Compiling regex v1.11.1
Compiling metrics v0.24.1
Compiling tracing-serde v0.2.0
Compiling quanta v0.12.5
Compiling ordered-float v4.6.0
Compiling rand_xoshiro v0.6.0
Compiling tracing-log v0.2.0
Compiling thread_local v1.1.8
Compiling route-recognizer v0.3.1
Compiling sketches-ddsketch v0.3.0
Compiling async-compression v0.4.22
Compiling aho-corasick v1.1.3
Compiling eyre v0.6.12
Compiling clap_lex v0.7.4
Compiling moka v0.12.10
Compiling clap_builder v4.5.34
Compiling opentelemetry-proto v0.28.0
Compiling jsonrpsee-server v0.24.9
Compiling opentelemetry-http v0.28.0
Compiling metrics-util v0.19.0
Compiling tracing-subscriber v0.3.19
Compiling uuid v1.16.0
Compiling jsonrpsee-proc-macros v0.24.9
Compiling alloy-rpc-types-engine v0.12.6
Compiling op-alloy-consensus v0.11.4
Compiling clap_derive v4.5.32
Compiling crossbeam-channel v0.5.14
Compiling tagptr v0.2.0
Compiling indenter v0.3.3
```

```
   Compiling op-alloy-rpc-types-engine v0.11.4
   Compiling clap v4.5.34
   Compiling tracing-opentelemetry v0.29.0
   Compiling opentelemetry-otlp v0.28.0
   Compiling tower-http v0.5.2
   Compiling dotenv v0.15.0
   Compiling tower-http v0.6.2
   Compiling rustls-webpki v0.103.1
   Compiling tokio-rustls v0.26.2
   Compiling hyper-rustls v0.27.5
   Compiling metrics-exporter-prometheus v0.16.2
   Compiling rustls-platform-verifier v0.5.1
   Compiling jsonrpsee-http-client v0.24.9
   Compiling jsonrpsee v0.24.9
   Compiling rollup-boost v0.1.0 (https://github.com/flashbots/rollup-boost.git?rev=eca9266#eca9266b)
   Compiling tx-proxy v0.1.0 (/Users/bogdanogorodniy/projects/audits/tx-proxy)
    Finished `release` profile [optimized] target(s) in 1m 28s
```

8.2 Tests Output

```
  tx-proxy git:(0fac2d2)  cargo test
   Compiling ctor v0.3.6
   Compiling tx-proxy v0.1.0 (/Users/bogdanogorodniy/projects/audits/tx-proxy)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 3.46s
     Running unittests src/lib.rs (target/debug/deps/tx_proxy-83a87101845e1841)

running 4 tests
test rpc::tests::test_parse_error_response_payload ... ok
test rpc::tests::test_parse_success_response_payload ... ok
test proxy::tests::test_send_raw_transaction_sad_path ... ok
test proxy::tests::test_send_raw_transaction_happy_path ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 2.78s

     Running unittests src/bin/main.rs (target/debug/deps/main-acdfd8e8124586d7)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests tx_proxy

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

8.3 Test coverage

Filename	Regions	Missed Reg.	Cover	Funcs	Missed F.	Exec.	Lines	Missed L.
bin/main.rs	6	6	0.00%	2	2	0.00%	11	11
cli.rs	121	121	0.00%	31	31	0.00%	199	199
client.rs	1	0	100.00%	1	0	100.00%	18	0
fanout.rs	10	1	90.00%	4	0	100.00%	17	0
proxy.rs	170	32	81.18%	24	0	100.00%	386	23
rpc.rs	42	5	88.10%	11	0	100.00%	107	0
validation.rs	35	4	88.57%	7	0	100.00%	49	0
TOTAL	385	169	56.10%	80	33	58.75%	787	233

Table 2: Code Coverage Report (cargo llvm-cov)

9 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