Security Review Report
NM-0493 World Chain Builder

NETHERMIND
SECURITY

(May 26, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for World Chain. This audit review focuses on the World Chain Builder, the block builder implementation for Priority Blockspace for Humans (PBH). PBH enables verified World ID users to execute transactions with top of block priority, enabling a more frictionless user experience. The mechanism is designed to ensure that users aren't disadvantaged by automated systems which greatly mitigates the negative impacts of MEV.

The World Chain Builder is a custom block builder for the OP Stack, featuring a priority blockspace available to PBH transactions, and standard blockspace for regular transactions. Transactions within the priority blockspace are executed first, with the standard blockspace being executed after. Users are entitled to a limited number of priority transactions per month, determined by the onchain ERC-4337 inspired `EntryPoint` contract.

The audit comprises 3161 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 10 points of attention, where two are classified as `Medium`, one is classified as `Low`, four are classified as `Informational`, and three are classified as `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.

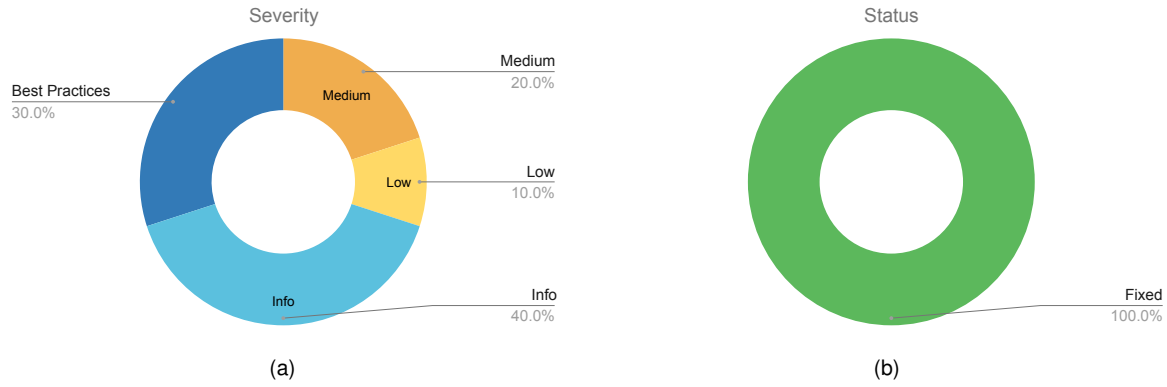

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (1), **Undetermined** (0), **Informational** (4), **Best Practices** (3). **Distribution of status: Fixed** (10), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 15, 2025
Response from Client	Regular responses during audit engagement
Final Report	May 26, 2025
Repositories	world-chain
Initial Commit	`f8eceb43b0553c27a4afe1ce2315d55c8489f250`
Final Commit	`99ff4a3afb75a9fedaa6e4a36d0e72777a9a5ddc`
Documentation	`pbh_tx_lifecycle.md`
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	crates/world/bin/src/main.rs	43	4	9.3%	6	53
2	crates/world/payload/src/lib.rs	1	1	100.0%	0	2
3	crates/world/payload/src/builder.rs	662	82	12.4%	77	821
4	crates/world/node/src/lib.rs	3	2	66.7%	2	7
5	crates/world/node/src/node.rs	359	45	12.5%	37	441
6	crates/world/node/src/args.rs	11	20	181.8%	6	37
7	crates/world/rpc/src/core.rs	36	10	27.8%	5	51
8	crates/world/rpc/src/transactions.rs	191	12	6.3%	26	229
9	crates/world/rpc/src/error.rs	14	6	42.9%	2	22
10	crates/world/rpc/src/lib.rs	8	1	12.5%	3	12
11	crates/world/rpc/src/sequencer.rs	105	13	12.4%	14	132
12	crates/world/pool/src/eip4337.rs	7	0	0.0%	3	10
13	crates/world/pool/src/error.rs	17	12	70.6%	2	31
14	crates/world/pool/src/lib.rs	19	4	21.1%	4	27
15	crates/world/pool/src/validator.rs	456	41	9.0%	107	604
16	crates/world/pool/src/tx.rs	220	8	3.6%	50	278
17	crates/world/pool/src/root.rs	186	54	29.0%	27	267
18	crates/world/pool/src/noop.rs	248	2	0.8%	46	296
19	crates/world/pool/src/ordering.rs	69	13	18.8%	15	97
20	crates/world/pool/src/bindings.rs	64	4	6.2%	10	78
21	crates/world/pbh/src/lib.rs	3	1	33.3%	1	5
22	crates/world/pbh/src/external_nullifier.rs	129	27	20.9%	31	187
23	crates/world/pbh/src/date_marker.rs	68	15	22.1%	17	100
24	crates/world/pbh/src/payload.rs	242	46	19.0%	48	336
	Total	**3161**	**423**	**13.4%**	**539**	**4123**

3 Summary of Issues

	Finding	Severity	Update
1	Block gas limit `FIXED_GAS` cost adjustment can be skipped	Medium	Fixed
2	PBH gas and nonce limits can become stale	Medium	Fixed
3	Dependencies contain known vulnerabilities	Low	Fixed
4	Inner transaction validation failure does not return early	Info	Fixed
5	Redundant transaction gas limit check during block building	Info	Fixed
6	The builder EOA must be adequately funded for blocks to be built	Info	Fixed
7	`TransactionConditional` validations can be reordered to improve performance	Info	Fixed
8	ExternalNullifierError is only partially used	Best Practices	Fixed
9	Incorrect/misleading comments	Best Practices	Fixed
10	Proof function `from_flat` should be used to convert from flat proof	Best Practices	Fixed

4 System Overview

The following is a description of each directory/crate introduced in the World Chain Builder repository in-scope for this audit:

- **`world-chain-builder`**:
  Contains the main entry point and CLI for the world chain node, handling the configuration and the initialization of the node.
- **`world-chain-builder-node`**:
  Core node functionality, including configuration/arguments for the World Chain node, which extends Optimism rollup functionality.
- **`world-chain-builder-payload`**:
  Manages block payload construction, including transaction execution and state management
- **`world-chain-builder-pbh`**:
  Handles creating all the necessary entries for the PBH Transaction Envelope. It specifically handles veryfing correct construction of the external nullifier and validates the payload.
- **`world-chain-builder-pool`**:
  Transaction pool implementation, managing verified and unverified transactions with priority inclusion logic for the PBH bundles.
- **`world-chain-builder-rpc`**:
  RPC interface implementation, including extensions for conditional transactions and World Chain-specific RPC methods.

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 [Medium] Block gas limit `FIXED_GAS` cost adjustment can be skipped

File(s): payload/src/builder.rs

Description: During the block building process, the external nullifiers in a PBH transaction are extracted and stored in a `HashSet` called `spent_nullifier_hashes`. The purpose of this set is to track all used nullifier hashes, and at the end of the block building process a final transaction can be crafted to mark these hashes as used on the `entrypoint` contract. If any duplicate nullifier hash is identified within the same block, then that transaction will be marked as invalid and it will be skipped. A snippet showing this logic is presented below:

```
if let Some(payloads) = pooled_tx.pbh_payload() {
    if info.cumulative_gas_used + tx.gas_limit() > verified_gas_limit {
        best_txs.mark_invalid(tx.signer(), tx.nonce());
        continue;
    }
    if payloads
        .iter()
        .any(|payload| !spent_nullifier_hashes.insert(payload.nullifier_hash))
    {
        best_txs.mark_invalid(tx.signer(), tx.nonce());
        invalid_txs.push(*pooled_tx.hash());
        continue;
    }
}
```

The result of calling `insert` will be `True` if the nullifier hash did not previously exist in the set, and `False` if the hash has already been inserted into the set before. For two separate transactions sharing the same nullifier hash this approach functions correctly, where the nullifier hash was correctly consumed by the first transaction and the second transaction with the duplicate hash was dropped. The nullifier hash should remain in the set because the first transaction has still properly executed and consumed it.

However, for a single transaction that has two duplicate nullifier hashes, the first insert into the set will be unique, but the second insert will flag as a duplicate so the transaction will be skipped. Even though the transaction was skipped and therefore was not executed, the nullifier hash from the first insert will remain in the set and be marked as used.

This can affect the dynamic block gas limit adjustments which are applied at the end of every transaction execution. There is a `FIXED_GAS` cost and a `COLD_SSTORE_GAS` cost which the builder logic accounts for, and reduces that amount from the block gas limit. The `FIXED_GAS` is only applied on the first PBH transaction, and the `COLD_SSTORE_GAS` is applied once for every nullifier hash in every PBH transaction. This logic is shown below:

```
if let Some(payloads) = pooled_tx.pbh_payload() {
    if spent_nullifier_hashes.len() == payloads.len() {
        block_gas_limit -= FIXED_GAS
    }
    block_gas_limit -= COLD_SSTORE_GAS * payloads.len() as u64;
}
```

If a PBH transaction with two duplicate external nullifier hashes is submitted with the highest priority, the block builder will attempt to execute it first. This will result in the execution failing due to duplicate hashes, but one of the nullifier hashes will still remain in `spent_nullifier_hashes`. The genuine PBH transaction will follow, and the condition required to apply the `FIXED_GAS` will be impossible to fulfill due to the single entry from the first transaction.

Since the reduction of `FIXED_GAS` on the `block_gas_limit` is meant to account for a portion of the cost of the final transaction by the block builder to spend nullifier hashes, by skipping this reduction it introduces a small margin of error where if the amount of gas consumed by regular transactions is at or near the unadjusted `block_gas_limit`, executing the final builder transaction may exceed the total amount of gas on the block which will lead to execution reverting. The execution of this final transaction is done in a different area than regular transactions, where a revert will cause the block building attempt to fail.

Recommendation(s): Consider adding a check to ensure that there are no duplicate nullifier hashes in the same transaction, before inserting into the `spent_nullifier_hashes` hash set.

Status: Fixed

Update from the client: Fixed in PR #235.

6.2 [Medium] PBH gas and nonce limits can become stale

File(s): pool/src/validator.rs

Description: The fields `max_pbh_nonce` and `max_pbh_gas_limit` represent the maximum number of PBH transactions per user per month, and the maximum gas limit for a PBH transaction respectively. The values for these fields are loaded from the storage state of the `entrypoint` contract, as shown below:

```
pub fn new(...) -> Result<...> {
    let state = inner.client().state_by_block_id(BlockId::latest())?;

    // Load these fields from the entrypoint storage state
    let max_pbh_nonce: u16 = ((state
        .storage(pbh_entrypoint, PBH_NONCE_LIMIT_SLOT.into())?
        .unwrap_or_default()
        >> PBH_NONCE_LIMIT_OFFSET)
        & MAX_U16)
        .to();
    let max_pbh_gas_limit: u64 = state
        .storage(pbh_entrypoint, PBH_GAS_LIMIT_SLOT.into())?
        .unwrap_or_default()
        .to();
    //...
}
```

These fields are read from the contract and only set one time during the initialization of the `WorldChainTransactionValidator`. The `entrypoint` smart contract exposes setter functions which are able to change the values at any time by the owner address. If the owner changes the PBH nonce limit or the PBH gas limit on the `entrypoint` contract, the onchain validations will become out-of-sync with the builder validations. This can cause PBH transactions which should be considered valid according to the `entrypoint` contract, to be considered as invalid by the builder which may contain stale data, or vice versa. As a result, PBH transactions may be dropped by the builder unexpectedly or the builder may accept transactions only to have them revert during onchain execution.

The function `on_new_head_block` does have a check which allows the `max_pbh_nonce` and `max_pbh_gas_limit` fields to be updated, but this can only occur when both fields are zero. Changing the values in the `entrypoint` contract from some non-zero value to another value will not result in the fields being updated.

```
fn on_new_head_block<B>(...) where ...
{
    // Will only trigger if both fields are zero
    // Cannot handle entrypoint fields being updated
    if self.max_pbh_gas_limit.load(Ordering::Relaxed) == 0
        && self.max_pbh_nonce.load(Ordering::Relaxed) == 0
    {
        // Try and fetch the max pbh nonce and gas limit
        if let Ok(state) = self.inner.client().state_by_block_id(...) {
            // Try update max pbh nonce
            if let Some(max_pbh_nonce) = state.storage(...).ok().flatten(){...}
            // Try update pbh gas limit
            if let Some(max_pbh_gas_limit) = state.storage(...).ok().flatten(){...}
        }
    }
    //...
}
```

Recommendation(s): Consider checking the `max_pbh_nonce` and `max_pbh_gas_limit` upon every new block to ensure that the PBH requirements specified in the `entrypoint` contract remain consistent with the PBH requirements tracked in the builder.

Status: Fixed

Update from the client: Fixed in PR #237.

6.3 [Low] Dependencies contain known vulnerabilities

File(s): `Cargo.toml`

Description: `cargo-audit` reported several dependencies having known vulnerabilities:

```
Version:  0.5.14
Title:    crossbeam-channel: double free on Drop
Date:     2025-04-08
ID:       RUSTSEC-2025-0024
URL:      https://rustsec.org/advisories/RUSTSEC-2025-0024
Solution: Upgrade to >=0.5.15
```

```
Version:  0.10.71
Title:    Use-After-Free in `Md::fetch` and `Cipher::fetch`
Date:     2025-04-04
ID:       RUSTSEC-2025-0022
URL:      https://rustsec.org/advisories/RUSTSEC-2025-0022
Solution: Upgrade to >=0.10.72
```

```
Version:  1.44.0
Warning:  unsound
Title:    Broadcast channel calls clone in parallel, but does not require `Sync`
Date:     2025-04-07
ID:       RUSTSEC-2025-0023
URL:      https://rustsec.org/advisories/RUSTSEC-2025-0023
```

Full log: cargo-audit-out.log

Recommendation(s): Consider updating `tokio`, `crossbeam-channel` and `openssl` to latest versions. Note that latter two are transitive dependencies, so consider either updating source dependencies (can be discovered via `cargo tree -invert <crate-name>`), or patching the dependency directly.

Status: Fixed

Update from the client: Fixed in PR #232.

6.4 [Info] Inner transaction validation failure does not return early

File(s): `pool/src/validator.rs`

Description: The function `WorldChainTransactionValidator::validate_pbh_bundle` has a comment indicating that it should return early if the generic `OpTransactionValidation` validation fails. However, after the transaction is validated there is no check on `tx_outcome` to return early if the outcome is invalid. By not returning early, the PBH validations are unnecessarily completed and a failure is only detected at the end of the function:

```rust
pub async fn validate_pbh_bundle(...) -> TransactionValidationOutcome<Tx> {
    // Ensure that the tx is a valid OP transaction and return early if invalid
    let mut tx_outcome = self.inner.validate_one(origin, tx.clone()).await;

    // An early return on invalid tx_outcome would be expected here

    // PBH validation logic here
    // ...
}
```

Recommendation(s): Consider adding a check to return early on an invalid `tx_outcome` to reduce unnecessary PBH validations and to ensure that comments align with the code.

Status: Fixed

Update from the client: Fixed in PR #229.

6.5 [Info] Redundant transaction gas limit check during block building

File(s): payload/src/builder.rs

Description: When a block is being built in `execute_best_transactions`, the function `is_tx_over_limits` is used to validate the transaction and block data limit, as well as the transaction's gas limit:

```
pub fn is_tx_over_limits(...) -> bool {
    // Verify tx data limit
    if tx_data_limit.is_some_and(|da_limit| tx.length() as u64 > da_limit) {
        return true;
    }
    // Verify block data limit
    if block_data_limit.is_some_and(...){
        return true;
    }
    // Verify tx gas limit
    self.cumulative_gas_used + tx.gas_limit() > block_gas_limit
}
```

Once this validation is passed, there is another gas limit check after the conditional transaction validation. This additional check is unnecessary since the `is_tx_over_limits` function already ensures that the transaction's gas will be within the specified limits.

```
// ensure we still have capacity for this transaction
if info.cumulative_gas_used + tx.gas_limit() > block_gas_limit {
    // we can't fit this transaction into the block, so we need to mark it as
    // invalid which also removes all dependent transaction from
    // the iterator before we can continue
    best_txs.mark_invalid(tx.signer(), tx.nonce());
    continue;
}
```

Recommendation(s): Consider removing the unnecessary transaction gas limit check.

Status: Fixed

Update from the client: Fixed in PR #230.

6.6 [Info] The builder EOA must be adequately funded for blocks to be built

File(s): payload/src/builder.rs

Description: After all PBH and regular transactions have been processed during the block building process, a final transaction is crafted which marks all relevant nullifier hashes as used. The signer for this transaction is a trusted EOA which has permissions to access privileged functions on the `entrypoint` contract.

```
pub fn spend_nullifiers_tx<...>(...) -> ... where ..., {
    // ...

    let mut tx = OpTransactionRequest::default()
        .nonce(nonce)
        .gas_limit(dyn_gas_limit(nullifier_hashes.len() as u64))
        .max_priority_fee_per_gas(evm.ctx().block.basefee.into())
        .max_fee_per_gas(evm.ctx().block.basefee.into())
        .with_chain_id(evm.ctx().cfg.chain_id)
        .with_call(&spendNullifierHashesCall {
            _nullifierHashes: nullifier_hashes.into_iter().collect(),
        })
        .to(ctx.pbh_entry_point)
        .build_typed_tx()
        .map_err(|e| eyre!("{:?}", e))?;

    // ...
}
```

This transaction has its gas pricing set to the current block's `basefee` which will be some nonzero amount means each transaction will incur a cost. Therefore, builder EOA addresses must always be adequately funded with native assets to successfully execute these hardcoded transactions. If a builder EOA does not have enough native assets for the transaction, the `execute_best_transactions` function will fail and blocks cannot be built until the builder EOA address has adequate funds again.

Recommendation(s): Ensure that the builder EOA address always has adequate funds to avoid block building issues. Alternatively, since these builder transactions are hardcoded to be included in the block regardless of gas priority, the gas price for the transaction could be set to zero and it would still be included in the block. This approach may lead to a simplification of the `block_gas_limit` dynamic adjustment based on the number of PBH nullifier hashes as well. In this case, please ensure that such a transaction won't fail validation on the `op-geth` node.

Status: Fixed

Update from the client: Fixed in PR #241. The builder should continue constructing a payload even if it can not submit a transaction to insert used PBH nullifier hashes into the PBH Entrypoint. This prevents a scenario where block production fails due to insufficient builder funds. By continuing with block construction, we ensure there are no slow downs, stuck transactions or negative impact on the user's ability to transact. Unspent nullifier hashes can be reused while the builder is underfunded. Proper monitoring, alerting, and automated funding should be in place to mitigate this.

6.7 [Info] `TransactionConditional` validations can be reordered to improve performance

File(s): rpc/src/transactions.rs

Description: The function `validate_conditional_options` is used during the block building process to determine if the conditions provided with the transaction can be satisfied. These checks can involve block timestamps, block numbers and storage state of contract addresses.

```
pub fn validate_conditional_options<...>(...) -> RpcResult<()> where ...,
{
    //...
    // Contract address storage state is checked first
    validate_known_accounts(
        &options.known_accounts,
        latest.header().number().into(),
        provider,
    )?;

    // Timestamps and block are checked after
    if let Some(min_block) = options.block_number_min {
        if min_block > latest.header().number() {
            return Err(ErrorCode::from(-32003).into());
        }
    }

    // More block and timestamp checks...
    // ...
}
```

Validating the storage state of contract addresses is much more expensive than timestamp and block related validations. By validating the block numbers and timestamps first, conditional transactions which have an invalid block number or timestamp can be dropped before reaching the more expensive storage state checks.

Recommendation(s): Consider changing the logic flow in `validate_conditional_options` to validate the inexpensive block numbers and timestamp checks before running the more expensive storage state checks.

Status: Fixed

Update from the client: Fixed in PR #231.

6.8 [Best Practices] ExternalNullifierError is only partially used

File(s): pbh/src/external_nullifier.rs

Description: The file `external_nullifier.rs` defines an error enum `ExternalNullifierError` which contains errors related to invalid external nullifier fields and parsing issues. Only one of the fields (`RlpError`) is used, with all other fields remaining unused.

```
#[derive(Debug, Clone, PartialEq, Eq, Error)]
pub enum ExternalNullifierError {
    #[error("invalid format: {0}")]
    InvalidFormat(#[from] ruint::ParseError),

    #[error("{0} is not a valid month number")]
    InvalidMonth(u8),

    #[error("error parsing external nullifier version")]
    InvalidVersion,

    #[error("error parsing external nullifier")]
    InvalidExternalNullifier,

    // Only this field is used
    #[error(transparent)]
    RlpError(#[from] alloy_rlp::Error),
}
```

In other functions defined in `external_nullifier.rs` the `alloy_rlp::Error::Custom` error is used instead, where some fields from the `ExternalNullifierError` might be more appropriate. For example the following code snippet could use the `ExternalNullifierError::InvalidExternalNullifier` instead of the custom error:

```
if version != Prefix::V1 as u8 {
    return Err(alloy_rlp::Error::Custom(
        "invalid external nullifier version",
    ));
}
```

Recommendation(s): To improve error consistency related to external nullifiers consider using all fields in `ExternalNullifierError` where applicable and removing unused errors. Alternatively, the `ExternalNullifierError` could be removed and `alloy_rlp` errors could be used for all errors.

Status: Fixed

Update from the client: Fixed in PR #238.

6.9 [Best Practices] Incorrect/misleading comments

File(s): node.rs

Description: The following is a list of comments which are misleading, incorrect or do not align with the logic in the codebase:

```
    L67  Comment refers to RollupArgs, should be WorldChainArgs
    L372 Comment refers to Optimism payload builder, should be WorldChain payload builder

payload/src/builder.rs
    L195 Comment refers to Optimism payload, should be WorldChain payload

rpc/src/transactions.rs
    L94  Debug string is cut off early, doesn't detail where raw transaction is sent to
```

Recommendation(s): Consider addressing the comments listed above.

Status: Fixed

Update from the client: Fixed in PR #239.

6.10 [Best Practices] Proof function `from_flat` should be used for flat proof

File(s): pool/src/bindings.rs

Description: The `try_from` implementation for `IPBHPayload` to `PBHPayload` manually extracts the flat proof data to construct a `Proof` type. However, the function `semaphore_rs::protocol::Proof::from_flat` already implements this logic and can be used instead:

```
fn try_from(val: IPBHPayload) -> Result<Self, Self::Error> {
    let g1a = (val.proof[0], val.proof[1]);
    let g2 = ([val.proof[2], val.proof[3]], [val.proof[4], val.proof[5]]);
    let g1b = (val.proof[6], val.proof[7]);

    let proof = Proof(semaphore_rs::protocol::Proof(g1a, g2, g1b));

    // This could be used instead
    let proof = from_flat(val.proof);

    Ok(PBHPayload {
        ...,
        proof,
    })
}
```

Recommendation(s): Consider using the library function provided by `Proof` directly.

Status: Fixed

Update from the client: Fixed in PR #240.

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. 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 World Chain Builder documentation

The **World Chain** team has provided documentation within the worldcoin/world-chain repository which provides information about the PBH implementation. The document pbh_tx_lifecycle.md details the motivation behind PBH, how priority transactions are created, the structure of PBH data, how transactions are validated and how transaction priority is determined.

Further questions and concerns raised by the Nethermind team were addressed through both meetings and async communication, providing valuable insight and a comprehensive understanding into the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
user@machine world-chain % cargo test --workspace

Running tests/mod.rs (target/debug/deps/mod-273680a2434f0605)
test test_invalidate_dup_tx_and_nullifier ... ok
test test_dup_pbh_nonce ... ok
test test_transaction_pool_ordering ... ok
test test_can_build_pbh_payload ... ok
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 12.54s

Running unittests src/lib.rs (target/debug/deps/world_chain_builder_pbh-c22eec7f63a7748d)
test date_marker::tests::parse_month_marker_invalid::empty ... ok
test date_marker::tests::parse_month_marker_invalid::invalid_month ... ok
test date_marker::tests::parse_month_marker_invalid::too_long ... ok
test date_marker::tests::parse_month_marker_invalid::too_short ... ok
test date_marker::tests::parse_month_marker_invalid::zero_month ... ok
test date_marker::tests::parse_month_marker_roundtrip::_012024_expects ... ok
test date_marker::tests::parse_month_marker_roundtrip::_022024_expects ... ok
test date_marker::tests::parse_month_marker_roundtrip::_022025_expects ... ok
test external_nullifier::tests::parse_external_nulliifer_roundtrip::externalnullifier_v1_1_2025_11_expects ... ok
test external_nullifier::tests::parse_external_nulliifer_roundtrip::externalnullifier_v1_12_3078_19_expects ... ok
test external_nullifier::tests::rlp_roundtrip::externalnullifier_v1_12_3078_19_expects ... ok
test external_nullifier::tests::rlp_roundtrip::externalnullifier_v1_1_2025_11_expects ... ok
test payload::test::invalid_external_nullifier_invalid_nonce ... ok
test payload::test::invalid_external_nullifier_invalid_period::_01_2024_0 ... ok
test payload::test::invalid_external_nullifier_invalid_period::_02_2025_0 ... ok
test payload::test::encode_decode ... ok
test payload::test::invalid_root ... ok
test payload::test::valid_external_nullifier::_01_2025_0 ... ok
test payload::test::valid_external_nullifier::_01_2025_1 ... ok
test payload::test::valid_external_nullifier::_01_2025_29 ... ok
test payload::test::valid_root ... ok
test payload::test::serialize_compressed_proof ... ok
test result: ok. 22 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 4.32s

Running unittests src/lib.rs (target/debug/deps/world_chain_builder_pool-5399734215c26ec5)
test ordering::test::higher_tip_has_priority::true_expects ... ok
test ordering::test::pbh_has_priority ... ok
test ordering::test::higher_tip_has_priority::false_expects ... ok
test validator::tests::validate_noop_non_pbh ... ok
test validator::tests::validate_no_duplicates ... ok
test root::tests::test_validate_root ... ok
test validator::tests::validate_bundle_no_pbh ... ok
test validator::tests::validate_date_marker_outdated ... ok
test validator::tests::validate_date_marker_in_the_future ... ok
test validator::tests::invalid_external_nullifier_nonce ... ok
test validator::tests::validate_pbh_bundle ... ok
test validator::tests::validate_pbh_multicall ... ok
test validator::tests::validate_pbh_bundle_missing_proof_for_user_op ... ok
test result: ok. 13 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 7.87s


Running unittests src/lib.rs (target/debug/deps/world_chain_builder_test_utils-a0cbeca5342bd337)
test utils::tests::test_pbh_nonce_key ... ok
test utils::tests::test_get_safe_op_hash ... ok
test utils::tests::mnemonic_accounts::_4_0x15d34aaf54267db7d7c367839aaf71a00a2c6a65_expects ... ok
test utils::tests::mnemonic_accounts::_3_0x90f79bf6eb2c4f870365e785982e1f101e93b906_expects ... ok
test utils::tests::mnemonic_accounts::_1_0x70997970c51812dc3a010c7d01b50e0d17dc79c8_expects ... ok
test utils::tests::mnemonic_accounts::_0_0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266_expects ... ok
test utils::tests::mnemonic_accounts::_8_0x23618e81e3f5cdf7f54c3d65f7fbc0abf5b21e8f_expects ... ok
test utils::tests::mnemonic_accounts::_5_0x9965507d1a55bcc2695c58ba16fb37d819b0a4dc_expects ... ok
test utils::tests::mnemonic_accounts::_2_0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc_expects ... ok
test utils::tests::mnemonic_accounts::_7_0x14dc79964da2c08b23698b3d3cc7ca32193d9955_expects ... ok
test utils::tests::mnemonic_accounts::_9_0xa0ee7a142d267c1f36714e4a8f75612f20a79720_expects ... ok
test utils::tests::mnemonic_accounts::_6_0x976ea74026e726554db657fa54763abd0c3a0aa9_expects ... ok
test utils::tests::treeroot ... ok
test result: ok. 13 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.21s
```

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

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.