
Security Review Report

NM-0417 Worldcoin Worldchain Contracts



NETHERMIND
SECURITY

(February 11, 2025)

Contents

| | | |
|-----------|--|-----------|
| 1 | Executive Summary | 2 |
| 2 | Audited Files | 3 |
| 3 | Summary of Issues | 3 |
| 4 | System Overview | 4 |
| 4.1 | PBH EntryPoint | 5 |
| 4.2 | PBH Signature Aggregator | 5 |
| 4.3 | PBH4337 Module | 5 |
| 5 | Risk Rating Methodology | 6 |
| 6 | Issues | 7 |
| 6.1 | [Medium] Nullifier hashes may be consumed without a valid proof | 7 |
| 6.2 | [Low] Missing length check for pbhPayloads and userOps | 8 |
| 6.3 | [Best Practices] Unnecessary gas check in pbhMulticall | 8 |
| 6.4 | [Best Practices] Opportunity to save gas in checkSignatures on next Safe release | 9 |
| 6.5 | [Best Practices] Natspec and comments | 9 |
| 7 | Documentation Evaluation | 10 |
| 8 | Slither | 10 |
| 9 | AuditAgent | 10 |
| 10 | Test Suite Evaluation | 11 |
| 11 | About Nethermind | 12 |

1 Executive Summary

This document outlines the security review conducted by [Nethermind Security](#) for the [World Chain](#) Priority Blockspace for Humans (PBH) smart contracts. PBH allows World ID users to execute transactions with top-of-block priority, providing a smoother and more seamless user experience. This functionality is powered by a custom block builder and EIP-4337-style smart contracts, which identify transactions containing valid World ID proofs. The priority blockspace is a fixed-size subset of the regular blockspace, with transactions in the priority blockspace being processed first, followed by regular transactions.

This audit engagement focuses only on the smart contract layer, which consists of three contracts:

- 1) A custom PBH entrypoint contract with World ID proof checking capabilities.
- 2) An EIP-4337 signature aggregator, which is tightly integrated with the PBH entrypoint.
- 3) A modified Safe Account EIP4337 module able to handle the unique PBH signature scheme.

The audited code comprises of 860 lines of code written in the Solidity language, and the audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract.

Along this document, we report five points of attention, where one is classified as Medium, one is classified as Low, and three are classified as Informational or 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 describes 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 and 9 detail automated tooling used during the audit. Section 10 presents the compilation, tests, and automated tests. Section 11 concludes the document.

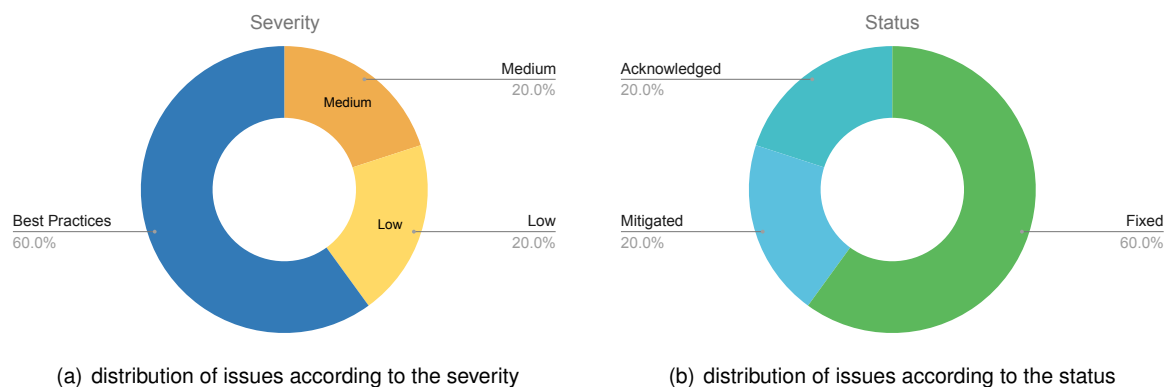


Fig 1: (a) Distribution of issues: Critical (0), High (0), Medium (1), Low (1), Undetermined (0), Informational (0), Best Practices (3). (b) Distribution of status: Fixed (3), Acknowledged (1), Mitigated (1), Unresolved (0)

Summary of the Audit

| | |
|---------------------------------|--|
| Audit Type | Security Review |
| Draft Report | January 28, 2025 |
| Final Report | February 11, 2025 |
| Methods | Manual Review, Automated analysis |
| Repository | worldcoin/world-chain |
| Initial Commit Hash | cc112c40157eca93e998649459340f4c5f3580c0 |
| Final Commit Hash | 64fa073799ffff9995bc6e415ef31c992113dff7 |
| Documentation Assessment | High |
| Test Suite Assessment | High |

2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|------------|------------|--------------|------------|------------|
| 1 | contracts/src/PBH4337Module.sol | 56 | 49 | 87.5% | 24 | 129 |
| 2 | contracts/src/PBHEntryPoint.sol | 6 | 21 | 350.0% | 4 | 31 |
| 3 | contracts/src/PBHEntryPointImplV1.sol | 152 | 97 | 63.8% | 49 | 298 |
| 4 | contracts/src/PBHSignatureAggregator.sol | 54 | 43 | 79.6% | 17 | 114 |
| 5 | contracts/src/lib/PBHExternalNullifier.sol | 32 | 39 | 121.9% | 10 | 81 |
| 6 | contracts/src/lib/SafeModuleSignatures.sol | 43 | 21 | 48.8% | 9 | 73 |
| 7 | contracts/src/lib/ByteHasher.sol | 6 | 5 | 83.3% | 1 | 12 |
| 8 | contracts/src/interfaces/IMulticall3.sol | 50 | 1 | 2.0% | 20 | 71 |
| 9 | contracts/src/interfaces/IPBHEntryPoint.sol | 33 | 6 | 18.2% | 12 | 51 |
| | Total | 432 | 282 | 65.3% | 146 | 860 |

3 Summary of Issues

| | Finding | Severity | Update |
|---|--|----------------|--------------|
| 1 | Nullifier hashes may be consumed without a valid proof | Medium | Unresolved |
| 2 | Missing length check for pbhPayloads and userOps | Low | Fixed |
| 3 | Unnecessary gas check in pbhMulticall | Best Practices | Fixed |
| 4 | PBH4337Module uses outdated checkSignatures function | Best Practices | Acknowledged |
| 5 | Natspec and comments | Best Practices | Fixed |

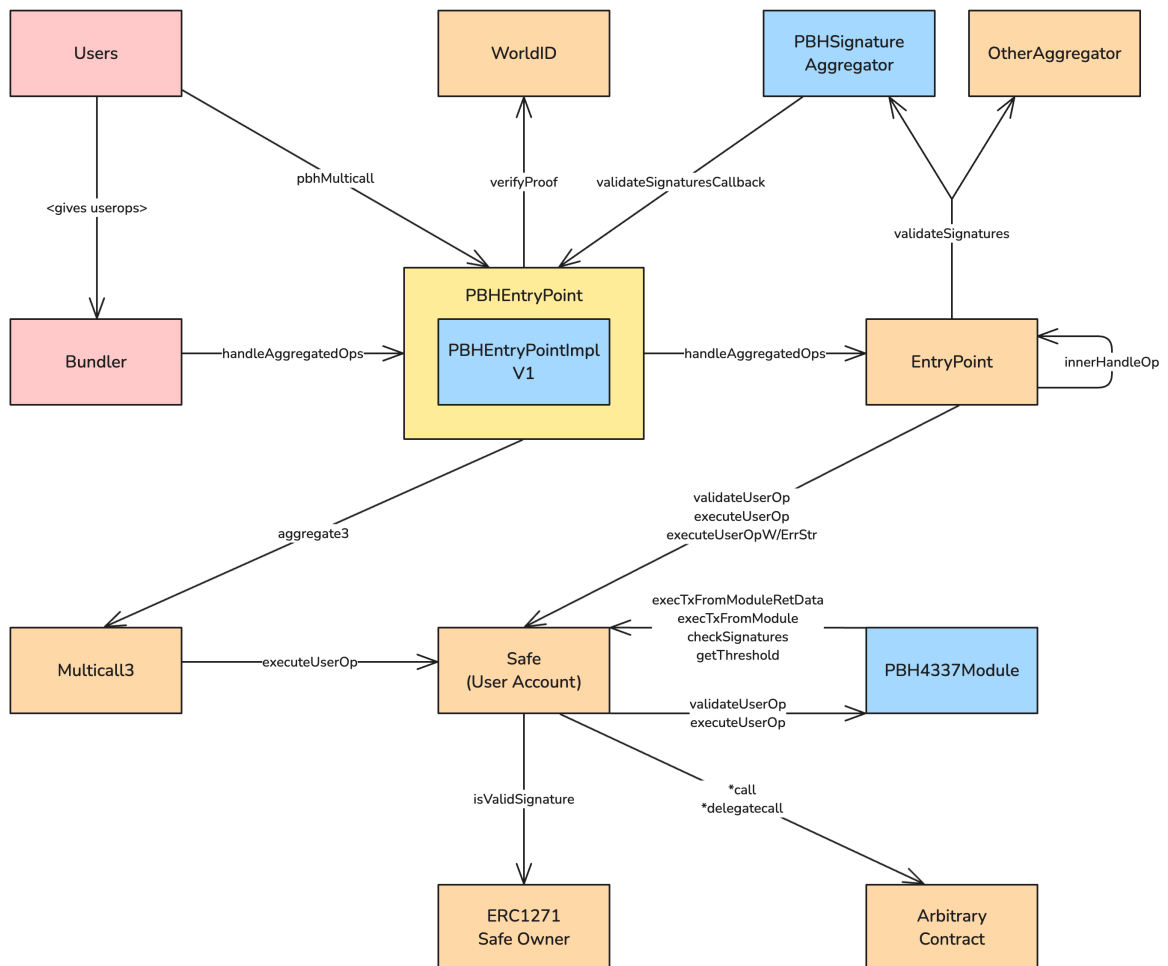
4 System Overview

The Worldcoin team introduced a custom block builder: World Chain Builder, which provides Priority Blockspace for Humans (PBH). With PBH a verified World ID user can benefit from top-of-block inclusion by providing a valid proof as part of their transaction. This mechanism ensures that certain transactions are prioritized for inclusion in the next block based on verified proof, offering a reliable way for users to access high-priority block space. The builder is designed to allocate a limited blockspace for PBH transactions per block, ensuring fair allocation between priority and non-priority transactions.

The PBH system consists of the following key components:

- **Bundlers:** Bundlers are off-chain entities responsible for collecting and submitting user operations (userOps) for PBH inclusion. A bundler groups multiple userOps together with an aggregator and submits the resulting bundles to the PBH EntryPoint.
- **PBH EntryPoint:** The PBHEntryPointImplV1 contract is the core contract responsible for PBH validation. It exposes the primary entry points for submitting PBH transactions. The PBHEntryPoint is a proxy contract that points to the PBHEntryPointImplV1 implementation, forwarding calls to the actual implementation.
- **Signature Aggregator:** The signature aggregator is a utility contract used by bundlers to group multiple userOps. It provides functions for signature validation, and is tightly integrated with the PBH endpoint contract, using transient storage to verify that a set of userOps already have valid proofs associated with them.
- **PBH4337 Module:** The PBH4337Module extends the Safe 4337 module, enabling the acceptance of a threshold of ECDSA or ERC1271 signatures, with or without an attached proof.

The following diagram illustrates the system's main actors and components, along with their interactions:



Worldchain Contracts Overview

4.1 PBH EntryPoint

Users can submit PBH transactions using one of two available methods. A PBH transaction can either contain a single-user operation or a bundle of multiple-user operations.

- **handleAggregatedOps**: The user submits a PBH user operation, or a bundle of user operations, to the builder, which invokes the `handleAggregatedOps` function to submit the PBH transaction. The associated proof data is either verified on-chain, through the WorldID's `verifyProof` function, or by the block builder itself. Each set of user operations associated with the same aggregator is hashed and stored in transient storage for later signature validation. Once the verification is complete, the call is forwarded to the actual entry point of the Safe 4337 module, which processes the user operations, validating their signatures and executing them.

```
function handleAggregatedOps(
    IEntryPoint.UserOpsPerAggregator[] calldata opsPerAggregator,
    address payable beneficiary
) external virtual onlyProxy onlyInitialized nonReentrant
```

- **pbhMulticall**: Users are able to execute multicalls with top-of-block inclusion through the `pbhMulticall` function.

```
function pbhMulticall(
    IMulticall3.Call3[] calldata calls,
    PBHPayload calldata pbhPayload
) external virtual onlyProxy onlyInitialized nonReentrant returns (
    IMulticall3.Result[] memory returnData
)
```

4.2 PBH Signature Aggregator

The `PBHSigntureAggregator` contract is used as a utility by the bundler to aggregate `UserOperation` proofs onto the signature to be passed to the `handleAggregatedOps` function. This aggregation is implemented through the `aggregateSignatures` function, which consolidates the signatures from the user operations into a single aggregated signature.

```
function aggregateSignatures(
    PackedUserOperation[] calldata userOps
) external view returns (bytes memory aggregatedSignature)
```

The `PBHSigntureAggregator` contract also defines the `validateSignatures` function, which is invoked by the Safe 4337 EntryPoint contract to validate the signatures of the user operations. This function forwards the call to the `PBHEntropyPoint`'s `validateSignaturesCallback` function. In this callback, the stored hashed user operations are compared with those provided. If they match, the signature is considered valid. This ensures that only correctly signed and verified operations are executed.

```
function validateSignatures(
    PackedUserOperation[] calldata userOps,
    bytes calldata
) external view
```

4.3 PBH4337 Module

The `PBH4337Module` extends the Safe 4337 module, allowing the handling of a set of signatures following the PBH signature schema. Eventually, it overrides the `_validateSignatures` function to allow the acceptance of a threshold of ECDSA signatures, with or without an attached proof.

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] Nullifier hashes may be consumed without a valid proof

File(s): PBHEntryPointImplV1.sol

Description: When attempting to execute a priority transaction through the PBH endpoint, a user must provide a valid external nullifier and nullifier hash. These are combined with other data to verify a proof, ensuring that the user has a valid World ID. To prevent replay attacks, a storage mapping nullifierHashes tracks all previously used hashes. Upon successful execution of a PBH transaction, the nullifier hash will be marked as "used" and cannot be used again.

The endpoint contract supports two PBH flows: handleAggregatedOps and pbhMulticall. Both functions verify the proof and mark the nullifier hash as used. If the endpoint storage variable worldId is not set, then proofs are not verified onchain, and it becomes the builder's responsibility to validate them.

```
function verifyPbh(uint256 signalHash, PBHPayload memory pbhPayload) ... {
    // Check for nullifier hash reuse
    if (nullifierHashes[pbhPayload.nullifierHash]) {
        revert InvalidNullifier();
    }

    // Verify the external nullifier
    PBHExternalNullifier.verify(pbhPayload.pbhExternalNullifier, numPbhPerMonth);

    // Skip onchain verification if `worldId` is not set
    if (address(worldId) != address(0)) {
        worldId.verifyProof(
            ..., pbhPayload.nullifierHash, ...
        );
    }
}
```

When the worldId is not set, it may be possible for a user to call either of the two endpoint functions with an invalid proof and still consume a nullifier hash. Depending on the builder implementation, there are two behaviors when an invalid proof is provided to the PBH endpoint contract:

1. Allow the transaction to be included in the block but without priority
2. Reject the transaction, preventing it from being included in the block

If the builder follows the first behavior, users could call handleAggregatedOps or pbhMulticall with invalid proofs and still consume these nullifier hashes. During discussions related to this finding, the Worldcoin team raised the concern that an entity could become a bundler and get access to user PBH calls before they have been executed. This malicious entity could then attempt to mark a user's nullifier hashes as used before the user transaction executes, causing the user's call to revert since the nullifier hash is already used.

Recommendation(s): Ensure that the Worldchain block builder excludes transactions that include a call to the PBH endpoint functions handleAggregatedOps and pbhMulticall if they do not meet the PBH conditions (invalid proof).

Status: Mitigated

Update from the client: This will be addressed by changing the PBH transaction inclusion criteria in the builder logic.

6.2 [Low] Missing length check for pbhPayloads and userOps

File(s): [PBHEntryPointImplV1.sol](#)

Description: In the function `handleAggregatedOps`, there is a nested loop which iterates through the `userOps` for each aggregator. Each `userOp` should have a `pbhPayload` associated with it, which is used in `verifyPbh` to ensure that a valid proof has been provided. If any `userOp` has an invalid proof then the transaction will revert. The code is shown below:

```
function handleAggregatedOps(...) ... {
    for (uint256 i = 0; i < opsPerAggregator.length; ++i) {
        //...
        PBHPayload[] memory pbhPayloads = abi.decode(
            opsPerAggregator[i].signature,
            (PBHPayload[])
        );
        for (uint256 j = 0; j < pbhPayloads.length; ++j) {
            address sender = opsPerAggregator[i].userOps[j].sender;
            // Generate signal hash for verification
            uint256 signalHash = abi.encodePacked(...).hashToField();
            // Verify proof
            verifyPbh(signalHash, pbhPayloads[j]);
            // ...
        }
    }
    // ...
}
```

However, there are no checks to ensure that the length of `pbhPayloads` extracted from the aggregator signature is equal to the number of `userOps` in the aggregator's struct. The inner loop which does the proof verification will stop once it has reached the length of `pbhPayloads`. When `pbhPayloads` is less than `userOps` the loop will stop executing even though there are additional user operations which have not been verified. These skipped `userOps` will not be verified and their nullifier hash will not be consumed, but their call will still execute successfully.

This method of skipping proof verification can be done even when the `worldId` address is set and onchain proof verification is supposed to occur. Since this logic flow of skipping proof verification is unexpected, the builder implementation may not be written in such a way that can detect this, and may consider all `userOps` as PBH regardless. If this is the case, then any user may be able to append additional `userOps` without an associated proof and bypass the `numPbhPerMonth` limit.

Recommendation(s): Consider adding a check to `handleAggregatedOps` to ensure that the length of `pbhPayloads` is equal to `opsPerAggregator`.

Status: Fixed

Update from the client: Addressed in [PR #140](#).

6.3 [Best Practices] Unnecessary gas check in pbhMulticall

File(s): [PBHEntryPointImplV1.sol](#)

Description: The function `pbhMulticall` verifies a user's calls and `pbhPayload` before making a call to `Multicall3.aggregate3`, which executes the users intended actions. After these calls are executed, the following gas check is done:

```
function pbhMulticall(...) ... returns (...) {
    // Verify calls and proof
    // ...
    returnData = IMulticall3(multicall3).aggregate3(gas: pbhGasLimit)(calls);
    // ...

    // Check if pbh gas limit is exceeded
    if (gasleft() > pbhGasLimit) {
        revert GasLimitExceeded(gasleft());
    }
    // ...
}
```

The purpose of this check is to ensure that an excessive amount of gas hasn't been spent in the transaction, as it may affect other PBH users if excessive gas amounts reach the block limits. However, this check is ineffective as it's possible for gas to have been spent before the call to `pbhMulticall`. The check only ensures that there is some small amount of gas (`pbhGasLimit`) remaining after the call.

Recommendation(s): Consider removing the ineffective check, and explore options for a more complete gas spend check, potentially one that is integrated into the builder.

Status: Fixed

Update from the client: Addressed in [PR #139](#).

6.4 [Best Practices] Opportunity to save gas in checkSignatures on next Safe release

File(s): [PBHEntryPointImplV1.sol](#)

Description: The PBH4337Module is an override of a forked Safe4337Module with a change to `_validateSignatures` which allows it to handle the unique PBH signature structure. Once the `userOp.signature` has been processed properly (prepended and appended data extracted) the module calls `checkSignatures` on the user's Safe smart contract account, as shown below:

```
function _validateSignatures(...) ... {
    // ...
    // Get threshold
    uint256 threshold = ISafe(payable(userOp.sender)).getThreshold();
    // ...
    // Extract signature
    if (isPBH && userOp.signature.length > expectedLength) {
        if (userOp.signature.length - expectedLength != ENCODED_PROOF_BYTES) {
            revert InvalidProofSize();
        }
        signatures = userOp.signature[TIMESTAMP_BYTES:expectedLength];
    }
    // ...
    try ISafe(payable(userOp.sender)).checkSignatures(keccak256(operationData), operationData, signatures) {}
    catch {
        validSignature = false;
    }
    // ...
}
```

In Safe's latest code (ahead of the latest version 1.4.1-3), there are two versions of this function, each accepting different arguments:

```
checkSignatures(bytes32,bytes,bytes)
checkSignatures(address,bytes32,bytes)
```

A [comment](#) in the Safe implementation states that the `(bytes32,bytes,bytes)` function will only exist for compatibility reasons, and that the newer `(address,bytes32,bytes)` should be used instead. When the next Safe version is released, using the newer function will reduce gas costs since the unhashed `operationData` won't have to be sent as part of the call, only the hashed data.

Recommendation(s): No action needs to be taken now, this finding only serves to highlight the opportunity to reduce gas costs during signature verification when a future Safe version is released.

Status: Acknowledged

Update from the client: Acknowledged.

6.5 [Best Practices] Natspec and comments

File(s): [/*](#)

Description: The following is a list of Natspec and comment corrections:

```
`PBHEntryPointImplV1.initialize` the args `multicall3` and `pbhGasLimit` are missing natspec comments
`PBHEntryPointImplV1.initialize` @dev comment for `_worldId` has typo "address" -> "address"
`PBHEntryPointImplV1.PBHEntryPointImplInitialized` event arg `pbhGasLimit` missing natspec comment
`PBHExternalNullifier.encode` the arg `version` is missing a natspec comment
```

Recommendation(s): Consider correcting the comments mentioned above.

Status: Fixed

Update from the client: Addressed in [PR #142](#) and [PR #153](#).

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 smart contract's design and technical details. It includes information about the purpose of the contract, 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 smart contract. 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 smart contract. 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 smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about World Chain documentation

The documentation for the World Chain contracts is contained in two README files in the project's GitHub repository.

The first README, located in the root directory, provides a high-level overview of PBH, covering both the smart contracts and block-building architecture. Within the contracts directory, there is a README specifically detailing the smart contracts. This document explains the dependencies of the contracts (EIP-4337, Safe Accounts) and provides an explanation of each contract. For the entrypoint contract, the two primary functions, `handleAggregatedOps` and `pbhMulticall`, are also covered.

The information is presented in a very concise and technical manner, with well-written explanations and references where necessary. It also features a section explaining development dependencies and instructions on how to build and test the code.

Code comments are also of high quality, with explanations for many functions describing the purpose, context, and situations where the function will be called. Other than functions, some comments exist in other areas of the code where extra information is necessary, allowing readers to understand the codebase at a faster pace.

8 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

9 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at <https://app.auditagent.nethermind.io>.

10 Test Suite Evaluation

The test suite for the PBH contracts is extensive, covering public functions of the contracts as well as internal functions, as in the case of PBH4337Module, where a function override occurs. The fuzzing features of Foundry are used throughout the test suite to ensure that functions behave correctly with a wide range of inputs. For the PBHEntryPoint logic, testing extends not only to expected function behavior but also to initialization and proxy-related behaviors, such as blocking access to direct interactions with the implementation contract.

```
Ran 6 tests for test/PBHEntryPointImplV1Proxy.t.sol:PBHEntryPointImplV1ProxyTest
[PASS] test_handleAggregatedOps_RevertIf_NotProxy
[PASS] test_pbhMulticall_RevertIf_Uninitialized
[PASS] test_setNumPbhPerMonth_RevertIf_Uninitialized
[PASS] test_setWorldId_RevertIf_Uninitialized
[PASS] test_validateSignaturesCallback_RevertIf_NotProxy
[PASS] test_verifyPbh_RevertIf_NotProxy

Ran 3 tests for test/PBH4337Module.t.sol:PBH4337ModuleTest
[PASS] testInvalidSignature
[PASS] testValidSignature_WithProof
[PASS] testValidSignature_WithoutProof

Ran 10 tests for test/PBHEntryPointImplV1Init.t.sol:PBHEntryPointImplV1InitTest
[PASS] test_handleAggregatedOps_RevertIf_Uninitialized
[PASS] test_initialize
[PASS] test_initialize_RevertIf_AddressZero
[PASS] test_initialize_RevertIf_AlreadyInitialized
[PASS] test_initialize_RevertIf_InvalidNumPbhPerMonth
[PASS] test_pbhMulticall_RevertIf_Uninitialized
[PASS] test_setNumPbhPerMonth_RevertIf_Uninitialized
[PASS] test_setWorldId_RevertIf_Uninitialized
[PASS] test_validateSignaturesCallback_RevertIf_Uninitialized
[PASS] test_verifyPbh_RevertIf_Uninitialized

Ran 9 tests for test/PBHExternalNullifier.t.sol:PBHExternalNullifierTest
[PASS] testFuzz_decode
[PASS] testFuzz_encode
[PASS] testFuzz_encode_RevertIf_InvalidMonth
[PASS] testFuzz_verify
[PASS] testFuzz_verify_RevertIf_InvalidExternalNullifierMonth
[PASS] testFuzz_verify_RevertIf_InvalidExternalNullifierVersion
[PASS] testFuzz_verify_RevertIf_InvalidExternalNullifierYear
[PASS] testFuzz_verify_RevertIf_InvalidNullifierLeadingZeros
[PASS] testFuzz_verify_RevertIf_InvalidPbhNonce

Ran 14 tests for test/PBHEntryPointImplV1.t.sol:PBHEntryPointImplV1Test
[PASS] test_handleAggregatedOps
[PASS] test_handleAggregatedOps_EIP1271
[PASS] test_handleAggregatedOps_RevertIf_Reentrancy
[PASS] test_pbhMulticall
[PASS] test_pbhMulticall_RevertIf_GasLimitExceeded
[PASS] test_pbhMulticall_RevertIf_Reentrancy
[PASS] test_setNumPbhPerMonth
[PASS] test_setNumPbhPerMonth_RevertIf_InvalidNumPbhPerMonth
[PASS] test_setNumPbhPerMonth_RevertIf_NotOwner
[PASS] test_setWorldId
[PASS] test_setWorldId_RevertIf_NotOwner
[PASS] test_validateSignaturesCallback_RevertIf_IncorrectHashedOps
[PASS] test_verifyPbh
[PASS] test_verifyPbh_RevertIf_InvalidNullifier

Ran 6 tests for test/PBHSignatureAggregator.t.sol:PBHSignatureAggregatorTest
[PASS] testFuzz_AggregateSignatures
[PASS] testFuzz_AggregateSignatures_EIP1271Signature
[PASS] testFuzz_AggregateSignatures_VariableThreshold
[PASS] testFuzz_ExtractProof,uint8
[PASS] testFuzz_ValidateUserOpSignature,uint8
[PASS] test_AggregateSignatures_RevertIf_InvalidSignatureLength
```

11 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process <https://www.overleaf.com/project/65c0e737f41a29601bda5c48ss>, 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.