Security Review Report
NM-0411-0817-A-WorldId

NETHERMIND
SECURITY

(February 24, 2026)

Contents

1 Executive Summary

This document presents the results of the security review conducted by Nethermind Security for World's **World ID** smart contracts.

The World team has developed a set of smart contracts that implement a privacy-preserving identity management system. The system allows users to prove membership of their accounts within a Merkle tree without disclosing their specific account identifiers. It achieves this through the integration of a Binary Merkle Tree, Poseidon2 hash function, and a set of registry and verifier contracts that manage identities, credential issuers, and proof verification.

Among the changes reviewed, the protocol now integrates a Full-Storage Binary Merkle Tree (`FullStorageBinaryIMT`), which replaces the previous frontier-based `InternalBinaryIMT`. The new implementation stores every internal node on-chain, allowing the contract to read sibling hashes directly from storage during tree updates. This eliminates the need for callers to supply Merkle proofs, enabling atomic updates and a simplified contract interface.

Additionally, a delayed recovery agent mechanism was introduced. Previously, recovery address changes took effect immediately. The new design requires an authenticator to initiate the update, a configurable cooldown period (default 14 days) to elapse, and only then can the update be executed. During the cooldown window, any valid authenticator may cancel the pending change, providing a safeguard against unauthorized recovery agent modifications.

The audit comprises 1218 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 3 points of attention, where one is classified as `High`, one is classified as `Medium` and one is classified as `Low`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 presents the summary of issues. 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 test suite evaluation and automated tools used. Section 9 concludes the document.

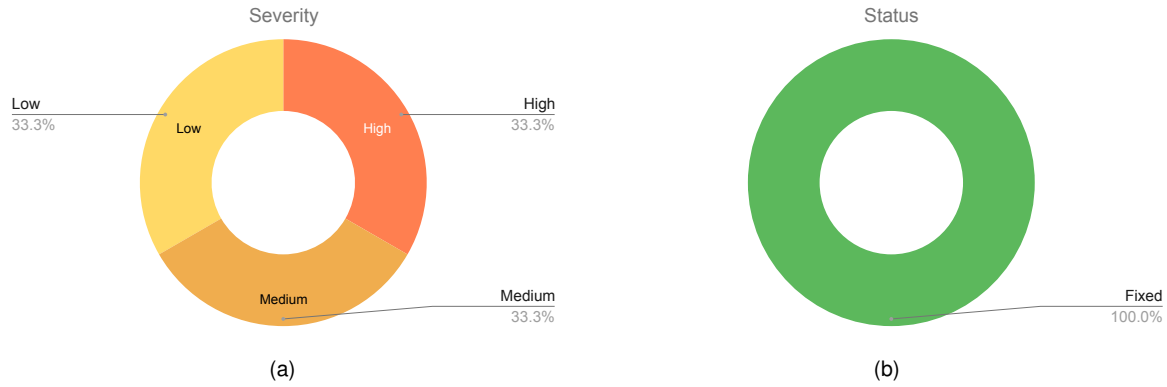

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (1), **Best Practices** (0). **Distribution of status: Fixed** (3), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	February 20, 2026
Final Report	February 24, 2026
Initial Commit	`0aad6`,`PR402`,`PR361`
Final Commit	`0b1bbb6`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	WorldIDRegistry.sol	534	147	27.5%	106	787
2	WorldIDVerifier.sol	197	60	30.5%	38	295
3	RpRegistry.sol	166	62	37.3%	53	281
4	CredentialSchemaIssuerRegistry.sol	225	65	28.9%	61	351
5	abstract/WorldIDBase.sol	81	80	98.8%	29	190
6	libraries/PackedAccountData.sol	15	38	253.3%	4	57
	Total	**1218**	**452**	**37.1%**	**291**	**1961**

In addition to the above files, the scope also includes the `PR #402` and `PR #361`.

3 Summary of Issues

	Finding	Severity	Update
1	The `updateRp(...)` function allows arbitrary `oprfKeyId` assignment without validation	High	Fixed
2	Decreasing `_maxAuthenticators` prevents management of previously registered authenticators	Medium	Fixed
3	The `removeAuthenticator(...)` function fails to validate the target authenticator's recovery counter allowing bitmap corruption	Low	Fixed

4 System Overview

World ID Protocol is a decentralized identity system that enables privacy-preserving proof-of-personhood. The core on-chain contracts manage identity registration, credential issuance, relying party registration, and zero-knowledge proof verification. All contracts follow a UUPS upgradeable proxy pattern with two-step ownership transfer, EIP-712 typed data signing, and an optional ERC-20 fee collection mechanism defined in a shared abstract base.

Each World ID account is represented as a leaf in an on-chain binary Merkle tree. Authenticators (on-chain signers) are authorized to manage accounts, while recovery agents provide a mechanism to fully reset the set of authenticators related to an account. Credential issuers and relying parties register through their respective registries, each of which initializes an OPRF key via the external `OprfKeyRegistry`.

4.1 WorldIDRegistry

The central registry for World ID accounts. Each account is identified by a `leafIndex` (starting at 1; index 0 is a sentinel) in a full-storage binary incremental Merkle tree (`FullStorageBinaryIMT`).

Core Data Structures:

```
// leafIndex -> [96 bits pubkeyId bitmap][160 bits recoveryAddress]
mapping(uint64 => uint256) internal _leafIndexToRecoveryAddressPacked;

// authenticatorAddress -> packed account data (leafIndex, recoveryCounter, pubkeyId)
mapping(address => uint256) internal _authenticatorAddressToPackedAccountData;
```

Each account supports up to `_maxAuthenticators` (default: 7, hard limit: 96) authenticators tracked via a 96-bit bitmap packed alongside the 160-bit recovery address in a single storage slot. Authenticator metadata is packed into a single `uint256` using the `PackedAccountData` library: `[32 bits recoveryCounter][32 bits pubkeyId][128 bits reserved][64 bits leafIndex]`.

Main Functions:

- `createAccount()` / `createManyAccounts()` – Registers one or multiple World ID accounts. Each account is assigned a leaf in the Merkle tree with an offchain signer commitment as the leaf value. Collects a registration fee if configured.
- `updateAuthenticator()` – Replaces an existing authenticator's on-chain address and public key. Requires an EIP-712 signature from the old authenticator, verified via ECDSA recovery. Updates the Merkle leaf with the new offchain signer commitment.
- `insertAuthenticator()` – Adds a new authenticator to an existing account at an unused `pubkeyId` slot. Requires a signature from any existing authenticator on the account.
- `removeAuthenticator()` – Removes an authenticator by clearing its bitmap bit and deleting its packed data. Requires a signature from any valid authenticator on the account.
- `recoverAccount()` – Account recovery via the designated recovery agent. Validates the signature using OpenZeppelin's `SignatureChecker` (supporting both EOA and ERC-1271 smart contract wallets). Increments the recovery counter (invalidating all previous authenticator mappings), resets the bitmap to a single authenticator, and clears any pending recovery agent update.
- `initiateRecoveryAgentUpdate()` – Begins a time-locked (default: 14 days) process to change the recovery agent. Signed by an existing authenticator. If a pending update already exists, it is overwritten (with a cancellation event emitted).
- `executeRecoveryAgentUpdate()` – Finalizes the recovery agent change after the cooldown period. Permissionless (no signature required).
- `cancelRecoveryAgentUpdate()` – Cancels a pending recovery agent update. Requires a signature from a valid authenticator.

Root Validity: Merkle roots are tracked with timestamps and remain valid for `_rootValidityWindow` seconds (default: 3600). The latest root is always valid.

Replay Protection: All signature-based operations use a per-account monotonically increasing nonce (`_leafIndexToSignatureNonce`). Additionally, the recovery counter mechanism ensures that after an account recovery, all pre-recovery authenticator mappings become stale, as their embedded recovery counter no longer matches the account's current counter.

4.2 CredentialSchemaIssuerRegistry

Registry for credential issuers. Each issuer registers a unique `issuerSchemaId` representing the combination of a specific credential schema and a specific issuer entity. Registration initializes an OPRF key via the external `OprfKeyRegistry` using `uint160(issuerSchemaId)` as the key ID.

Core Data Structures:

```
mapping(uint64 => Pubkey) internal _idToPubkey;    // off-chain signing key (x, y)
mapping(uint64 => address) internal _idToSigner;   // on-chain authorized signer
mapping(uint64 => uint256) internal _idToSignatureNonce; // replay protection
mapping(uint64 => string) internal _idToSchemaUri;  // schema definition URI
```

Main Functions:

- `register()` – Registers a new issuer-schema pair with an off-chain public key and an on-chain signer. Collects a fee if configured. Calls `_oprfKeyRegistry.initKeyGen()` to initialize OPRF key generation. The `issuerSchemaId` must be globally unique (also unique across the `RpRegistry` in the `OprfKeyRegistry`).
- `remove()` – Removes an issuer-schema pair. Requires an EIP-712 signature from the authorized signer. Deletes the public key, signer, and schema URI.
- `updatePubkey()` – Updates the off-chain credential-signing public key. Signature-authorized by the on-chain signer. Hashes both old and new pubkeys using the `PUBKEY_TYPEHASH` for EIP-712 structured data.
- `updateSigner()` – Rotates the on-chain signer authorized to manage the issuer-schema pair. Must be signed by the current signer.
- `updateIssuerSchemaUri()` – Updates the schema URI. Reverts if the new URI matches the current one.

Owner Functions: `updateOprfKeyRegistry()` allows the contract owner to update the external OPRF key registry address.

4.3 RpRegistry

Registry for Relying Parties (RPs) – entities that request World ID proofs from users. Each RP is identified by a unique `rpId` and stores a `RelyingParty` struct:

```
struct RelyingParty {
    bool initialized;
    bool active;
    address manager;     // authorized to manage RP record
    address signer;      // authorized to sign proof requests
    uint160 oprfKeyId;   // OPRF key ID (= uint160(rpId))
    string unverifiedWellKnownDomain; // FQDN for metadata
}
```

Main Functions:

- `register()` / `registerMany()` – Registers one or multiple RPs. Each registration initializes an OPRF key via the external registry using `uint160(rpId)` as the key ID, collects a fee if configured, and stores the RP record as active.
- `updateRp()` – Partially updates an RP record. Requires an EIP-712 signature from the current manager. Fields use sentinel values to indicate "no change": `address(0)` for manager/signer means no update, and the string `"__NO_UPDATE__"` for the domain means no update. The `toggleActive` boolean flips the active status when set to `true`.

Owner Functions: `updateOprfKeyRegistry()` allows the contract owner to update the external OPRF key registry address.

4.4 WorldIDVerifier

The verification contract that ties all registries together. It validates World ID zero-knowledge proofs by cross-referencing public inputs against the `WorldIDRegistry`, `CredentialSchemaIssuerRegistry`, and `OprfKeyRegistry`, then delegates the actual Groth16 proof verification to a `Verifier` contract.

Main Functions:

- `verify()` – Verifies a Uniqueness Proof. Sets `sessionId` to 0 and delegates to `verifyProofAndSignals()`.
- `verifySession()` – Verifies a Session Proof. Uses a `sessionNullifier` tuple and a non-zero `sessionId`.
- `verifyProofAndSignals()` – The core verification logic. Validates the Merkle root against the `WorldIDRegistry`, retrieves the issuer's public key from `CredentialSchemaIssuerRegistry`, retrieves the OPRF public key from `OprfKeyRegistry`, enforces the minimum expiration threshold (i.e., `expiresAtMin + _minExpirationThreshold >= block.timestamp`), then assembles 15 public signals and delegates to the `Verifier` contract's `verifyCompressedProof()`.

Owner Functions: The owner can update all four external contract references (`CredentialSchemaIssuerRegistry`, `WorldIDRegistry`, `OprfKeyRegistry`, `Verifier`) and the `_minExpirationThreshold`.

Replay Protection: Replay protection is intentionally enforced by the RP and not by this contract. Since the zero-knowledge proof is not bound to `msg.sender`, and the nonce and nullifier are not nullified within this contract, proofs may otherwise be replayed. Therefore, the calling contract is responsible for implementing the necessary checks to enforce nonce and nullifier nullification and to ensure that proofs cannot be reused.

4.5 Libraries

- **PackedAccountData** – Packs and unpacks authenticator metadata into a single `uint256: [32 bits recoveryCounter][32 bits pubkeyId][128 bits reserved][64 bits leafIndex]`. Provides `pack()`, `leafIndex()`, `recoveryCounter()`, and `pubkeyId()` helpers.

- **FullStorageBinaryIMT** – A full-storage binary incremental Merkle tree (up to depth 30) that persists all internal nodes in a `mapping(uint256 => uint256)` keyed by `(level « 32) | index`. Because siblings are always readable from storage, leaf updates do not require caller-supplied proofs. Supports `insert()`, `insertMany()`, `update()`, `remove()`, `verify()`, and `getProof()`. Uses Poseidon2 as the hash function. Pre-computes default zero values as constants for each level.

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] The `updateRp(...)` function allows arbitrary `oprfKeyId` assignment without validation

File(s): `contracts/src/RpRegistry.sol`

Description: When a Relying Party (RP) is first registered via `_register`, the contract derives its `oprfKeyId` deterministically as `uint160(rpId)` and initiates a distributed key generation by calling `_oprfKeyRegistry.initKeyGen(oprfKeyId)`. This process establishes a critical trust binding between the RP and its specific cryptographic key.

However, the `updateRp` function allows the RP manager to overwrite the stored `oprfKeyId` with any non-zero `uint160` value provided in the call arguments. This assignment occurs without verifying if the new ID corresponds to a valid, initialized key, nor does it check for ownership or uniqueness.

```
function updateRp(/* ... */, uint160 oprfKeyId, /* ... */) external {
    // ...
    if (oprfKeyId != 0) {
        // @audit-issue This updates the oprf key id without restarting the key generation.
        _relyingParties[rpId].oprfKeyId = oprfKeyId;
    }
    // ...
}
```

Consequently, an RP manager can point their `oprfKeyId` to a key that was never generated, or hijack an ID belonging to another RP or Credential Schema Issuer. This breaks the expected 1:1 relationship between an entity and its OPRF key and bypasses the required key generation ceremony, potentially compromising the integrity of the OPRF verification system.

Recommendation(s): Consider removing the ability to update the `oprfKeyId` within the `updateRp(...)` function. If key rotation or updates are a required feature, the process should strictly validate the new ID and ensure it corresponds to a valid key generation ceremony, mirroring the logic enforced during the initial register(...) call.

Status: Fixed

Update from the client: Fixed in PR 414.

6.2 [Medium] Decreasing `_maxAuthenticators` prevents management of previously registered authenticators

File(s): contracts/src/WorldIDRegistry.sol

Description: The `WorldIDRegistry` contract manages authenticator slots for users, where the total number of allowed slots is governed by the `_maxAuthenticators` state variable. When an authenticator is registered, it is assigned a specific `pubkeyId`, which essentially acts as an index within the range of 0 to the current `_maxAuthenticators - 1`. The contracrt owner can update this global limit at any time via the `setMaxAuthenticators(...)` function.

However, a logic flaw exists in how the contract handles existing authenticators when this limit is decreased. Both the `updateAuthenticator(...)` and `removeAuthenticator(...)` functions perform a strict bounds check against the current value of `_maxAuthenticators`.

```
function updateAuthenticator(uint256 pubkeyId, /* ... */) external {
    // @audit-issue If _maxAuthenticators was lowered, existing pubkeyIds may now be out of bounds.
    if (pubkeyId >= _maxAuthenticators) {
        revert PubkeyIdOutOfBounds();
    }
    // ...
}
```

If the owner reduces `_maxAuthenticators` to a value lower than the index of an already-registered authenticator, that authenticator becomes "stuck." Any attempt by the authenticator to update or remove itself will trigger the `PubkeyIdOutOfBounds()` revert, as the `pubkeyId` is now greater than or equal to the new, smaller `_maxAuthenticators` limit.

This check is inconsistent across the contract. Other operations that verify signatures or packed account data do not validate the `pubkeyId` against the current `_maxAuthenticators`. For instance, the `updateRecoveryAddress(...)` function allows these "out-of-bounds" authenticators to continue signing and performing recovery address update.

Recommendation(s): Consider removing the `pubkeyId >= _maxAuthenticators` check in `updateAuthenticator` and `removeAuthenticator`, allowing existing authenticators to continue updating or removing their credentials even if the global maximum limit is subsequently lowered.lowered.

Status: Fixed

Update from the client: Fixed in PR 412.

6.3 [Low] The `removeAuthenticator(...)` function fails to validate the target authenticator's recovery counter allowing bitmap corruption

File(s): contracts/src/WorldIDRegistry.sol

Description: The `WorldIDRegistry` contract manages authenticator slots using a bitmap, where each set bit corresponds to an occupied `pubkeyId`. When an account undergoes recovery via `recoverAccount(...)`, the account's `recoveryCounter` is incremented, and the bitmap is reset to track only the new recovery authenticator. Previously registered authenticators are not explicitly deleted from the `_authenticatorAddressToPackedAccountData` mapping; instead, they are logically invalidated because their stored `recoveryCounter` no longer matches the account's current counter.

The `removeAuthenticator(...)` function allows users to remove a specific authenticator. While it validates that the caller (the signer) has the current `recoveryCounter`, it fails to perform the same validation on the target authenticator being removed. It only checks that the target exists, belongs to the correct `leafIndex`, and matches the provided `pubkeyId`.

```
function removeAuthenticator(address authenticatorAddress, uint256 pubkeyId, /* ... */) external {
    // ...
    uint256 packedToRemove = _authenticatorAddressToPackedAccountData[authenticatorAddress];

    // @audit-issue Validates existence, but not if the authenticator is stale (old recovery epoch).
    if (packedToRemove == 0) {
        revert AuthenticatorDoesNotExist(authenticatorAddress);
    }
    // ...
    uint256 actualPubkeyId = PackedAccountData.pubkeyId(packedToRemove);
    if (actualPubkeyId != pubkeyId) {
        revert MismatchedPubkeyId(pubkeyId, actualPubkeyId);
    }

    // Delete authenticator
    delete _authenticatorAddressToPackedAccountData[authenticatorAddress];
    // @audit-issue This clears the bit for `pubkeyId` even if the target was a stale authenticator.
    _setPubkeyBitmap(leafIndex, _getPubkeyBitmap(leafIndex) & ~(1 << pubkeyId));
}
```

This omission allows a user to "remove" a stale authenticator from a previous recovery epoch. Since the stale authenticator likely shares a `pubkeyId` with a currently active authenticator (or a reserved slot), removing it clears the corresponding bit in the bitmap.

The impact is that the bitmap becomes desynchronized from the actual state of active authenticators. Specifically, if a stale authenticator with `pubkeyId` 0 is removed, the bit for `pubkeyId` 0 is cleared, even if a valid, active authenticator currently occupies that slot. This allows the user to register a duplicate authenticator at `pubkeyId` 0, breaking the one-authenticator-per-slot invariant and bypassing the `_maxAuthenticators` limit.

Recommendation(s): Consider adding a check within `removeAuthenticator(...)` to verify that the target authenticator's `recoveryCounter` matches the account's current `recoveryCounter`. If the target is stale, the transaction should either revert or only clear the storage mapping without modifying the active `pubkeyId` bitmap.

Status: Fixed

Update from the client: Fixed in PR 417.

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 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 ID contracts documentation

`World's` team provided an overview of the World ID System components during the kick-off call with a detailed explanation of the intended functionalities and use cases of the system. Moreover, the team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
$ forge compile
[] Compiling...
[] Compiling 95 files with Solc 0.8.30
[] Solc 0.8.30 finished in 88.23s
Compiler run successful!
```

8.2 Tests Output

```
Ran 9 tests for test/WorldIDVerifierUpgrade.sol:VerifierUpgradeTest
[PASS] test_CannotInitializeTwice() (gas: 24838)
[PASS] test_ImplementationCannotBeInitialized() (gas: 2239859)
[PASS] test_OnlyOwnerCanUpdate() (gas: 35132)
[PASS] test_OwnershipTransfer() (gas: 2308224)
[PASS] test_UpdateCredentialSchemaIssuerRegistry() (gas: 29323)
[PASS] test_UpdateOprfKeyRegistry() (gas: 29781)
[PASS] test_UpdateWorldIDRegistry() (gas: 116608)
[PASS] test_UpgradeFailsForNonOwner() (gas: 2282657)
[PASS] test_UpgradeSuccess() (gas: 2349394)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 5.64ms (1.22ms CPU time)

Ran 6 tests for test/CredentialSchemaIssuerRegistryUpgrade.t.sol:CredentialSchemaIssuerRegistryUpgradeTest
[PASS] test_CannotInitializeTwice() (gas: 86020)
[PASS] test_ImplementationCannotBeInitialized() (gas: 2899046)
[PASS] test_OwnerCannotRegisterWithoutUpgrade() (gas: 92875)
[PASS] test_OwnershipTransfer() (gas: 2907061)
[PASS] test_UpgradeFailsForNonOwner() (gas: 2880987)
[PASS] test_UpgradeSuccess() (gas: 3077080)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 9.42ms (6.25ms CPU time)

Ran 1 test for test/Verifier.t.sol:VerifierTest
[PASS] testVerifyNullifier() (gas: 295811)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.31ms (11.06ms CPU time)

Ran 5 tests for test/WorldIDRegistryUpgrade.t.sol:WorldIDRegistryUpgradeTest
[PASS] test_CannotInitializeTwice() (gas: 16944)
[PASS] test_ImplementationCannotBeInitialized() (gas: 3689191)
[PASS] test_OwnershipTransfer() (gas: 3881669)
[PASS] test_UpgradeFailsForNonOwner() (gas: 3854557)
[PASS] test_UpgradeSuccess() (gas: 4825893)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 18.08ms (10.76ms CPU time)

Ran 49 tests for test/RpRegistry.t.sol:RpRegistryTest
[PASS] testCannotRegisterDuplicateIdInOprfKeyRegistry() (gas: 179278)
[PASS] testCannotRegisterDuplicateRpId() (gas: 151657)
[PASS] testCannotRegisterManyWithInsufficientFee() (gas: 414497)
[PASS] testCannotRegisterManyWithMismatchedArrayLengths() (gas: 29319)
[PASS] testCannotRegisterManyWithMismatchedManagersLength() (gas: 27259)
[PASS] testCannotRegisterManyWithMismatchedSignersLength() (gas: 27678)
[PASS] testCannotRegisterRpWithIdThatConflictsWithCredentialRegistry() (gas: 56377)
[PASS] testCannotRegisterWithInsufficientFee() (gas: 135269)
[PASS] testCannotRegisterWithInvalidId() (gas: 24522)
[PASS] testCannotRegisterWithZeroAddressManager() (gas: 22064)
[PASS] testCannotRegisterWithZeroAddressSigner() (gas: 22057)
[PASS] testCannotSetFeeRecipientToZeroAddress() (gas: 18876)
[PASS] testCannotSetFeeTokenToZeroAddress() (gas: 19360)
[PASS] testCannotUpdateOprfKeyRegistryToZeroAddress() (gas: 19404)
[PASS] testInitialized() (gas: 18274)
[PASS] testMultipleRpsWithDifferentIds() (gas: 404754)
[PASS] testOnlyOwnerCanSetFeeRecipient() (gas: 31980)
[PASS] testOnlyOwnerCanSetFeeToken() (gas: 585232)
[PASS] testOnlyOwnerCanSetRegistrationFee() (gas: 19487)
[PASS] testOnlyOwnerCanUpdateOprfKeyRegistry() (gas: 144095)
[PASS] testOprfKeyIdMatchesRpId() (gas: 146499)
[PASS] testRegister() (gas: 146489)
```

```
[PASS] testRegisterMany() (gas: 388521)
[PASS] testRegisterManyInvalidId() (gas: 33409)
[PASS] testRegisterManyValidatesEachRegistration() (gas: 154331)
[PASS] testRegisterManyValidatesManagerNotZero() (gas: 154085)
[PASS] testRegisterManyValidatesSignerNotZero() (gas: 153902)
[PASS] testRegisterManyWithEmptyArrays() (gas: 15184)
[PASS] testRegisterManyWithFee() (gas: 495161)
[PASS] testRegisterMultipleRps() (gas: 267774)
[PASS] testRegisterWithExcessFee() (gas: 282354)
[PASS] testRegisterWithFee() (gas: 242856)
[PASS] testSetFeeRecipient() (gas: 31883)
[PASS] testSetFeeToken() (gas: 584255)
[PASS] testSetRegistrationFee() (gas: 45263)
[PASS] testUpdateOprfKeyRegistry() (gas: 143191)
[PASS] testUpdateRpCannotReplayOldSignature() (gas: 204224)
[PASS] testUpdateRpInvalidEIP1271Signature() (gas: 433169)
[PASS] testUpdateRpInvalidSignature() (gas: 173552)
[PASS] testUpdateRpManagerTransfer() (gas: 255396)
[PASS] testUpdateRpNonExistentRp() (gas: 32910)
[PASS] testUpdateRpNonceIncrementsOnEachUpdate() (gas: 237437)
[PASS] testUpdateRpNonceMismatch() (gas: 164011)
[PASS] testUpdateRpOprfKeyId() (gas: 204240)
[PASS] testUpdateRpPartialUpdate() (gas: 203643)
[PASS] testUpdateRpSuccess() (gas: 210753)
[PASS] testUpdateRpToggleActive() (gas: 232983)
[PASS] testUpdateRpWithEIP1271Signature() (gas: 460138)
[PASS] testUpdateRpWithNoUpdate() (gas: 203217)
Suite result: ok. 49 passed; 0 failed; 0 skipped; finished in 24.87ms (19.74ms CPU time)

Ran 6 tests for test/HashBench.t.sol:LeanIMTTest
[PASS] test_Poseidon2T2() (gas: 7360898)
[PASS] test_Poseidon2T2EqualsReference() (gas: 4743435)
[PASS] test_Poseidon2T2EqualsRust() (gas: 42730)
[PASS] test_Poseidon2T2Reference() (gas: 40115964)
[PASS] test_PoseidonT3() (gas: 42365716)
[PASS] test_PoseidonT4() (gas: 37701478)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 498.08ms (1.42s CPU time)

Ran 36 tests for test/CredentialSchemaIssuerRegistry.t.sol:CredentialIssuerRegistryTest
[PASS] testCannotRegisterDuplicateIdInCredentialRegistry() (gas: 120435)
[PASS] testCannotRegisterDuplicateIdInOprfKeyRegistry() (gas: 149230)
[PASS] testCannotRegisterWithEmptyPubkey() (gas: 38083)
[PASS] testCannotRegisterWithInsufficientFee() (gas: 132987)
[PASS] testCannotRegisterWithZeroSigner() (gas: 18087)
[PASS] testCannotReplayIssuerSchemaUri() (gas: 281271)
[PASS] testCannotSetFeeRecipientToZeroAddress() (gas: 18921)
[PASS] testCannotSetFeeTokenToZeroAddress() (gas: 19005)
[PASS] testCannotUpdateOprfKeyRegistryToZeroAddress() (gas: 19099)
[PASS] testCannotUpdateSchemaUriAfterRemoval() (gas: 146983)
[PASS] testCannotUpdateSchemaUriForNonExistentIssuer() (gas: 42519)
[PASS] testCannotUpdateSchemaUriToSameSchemaUri() (gas: 253700)
[PASS] testMultipleIssuersWithDifferentIds() (gas: 322284)
[PASS] testOnlyIssuerCanUpdateSchemaUri() (gas: 150697)
[PASS] testOnlyOwnerCanSetFeeRecipient() (gas: 30300)
[PASS] testOnlyOwnerCanSetFeeToken() (gas: 583485)
[PASS] testOnlyOwnerCanSetRegistrationFee() (gas: 17642)
[PASS] testOnlyOwnerCanUpdateOprfKeyRegistry() (gas: 168813)
[PASS] testRegisterAndGetters() (gas: 123942)
[PASS] testRegisterWithExcessFee() (gas: 254274)
[PASS] testRegisterWithFee() (gas: 214579)
[PASS] testRegisterWithZeroFee() (gas: 127757)
[PASS] testRemoveDeletesSchemaUri() (gas: 208015)
[PASS] testRemoveFlow() (gas: 138079)
[PASS] testRemoveWithERC1271Wallet() (gas: 359760)
[PASS] testSetFeeRecipient() (gas: 31735)
[PASS] testSetFeeToken() (gas: 584235)
[PASS] testSetRegistrationFee() (gas: 45003)
[PASS] testUpdateIssuerSchemaUriFlow() (gas: 277037)
[PASS] testUpdateIssuerSchemaUriWithERC1271Wallet() (gas: 504280)
[PASS] testUpdateOprfKeyRegistry() (gas: 171071)
[PASS] testUpdatePubkeyFlow() (gas: 177487)
[PASS] testUpdatePubkeyWithERC1271Wallet() (gas: 439394)
[PASS] testUpdateSignerFlow() (gas: 171498)
```

```
[PASS] testUpdateSignerToSameSigner() (gas: 133925)
[PASS] testUpdateSignerWithERC1271Wallet() (gas: 432147)
Suite result: ok. 36 passed; 0 failed; 0 skipped; finished in 498.49ms (19.01ms CPU time)

Ran 15 tests for test/WorldIDVerifierTest.t.sol:ProofVerifier
[PASS] test_CannotUpdateOprfKeyRegistryToZeroAddress() (gas: 18203)
[PASS] test_ExpiresAtTooOld() (gas: 60167)
[PASS] test_InvalidRoot() (gas: 47569)
[PASS] test_OnlyOwnerCanUpdateOprfKeyRegistry() (gas: 156829)
[PASS] test_SessionExpiresAtTooOld() (gas: 62623)
[PASS] test_SessionInvalidRoot() (gas: 49320)
[PASS] test_SessionSuccess() (gas: 357655)
[PASS] test_SessionWrongCredentialIssuer() (gas: 357954)
[PASS] test_SessionWrongProof() (gas: 347896)
[PASS] test_SessionWrongRpId() (gas: 358064)
[PASS] test_Success() (gas: 355431)
[PASS] test_UpdateOprfKeyRegistry() (gas: 161894)
[PASS] test_WrongCredentialIssuer() (gas: 356221)
[PASS] test_WrongProof() (gas: 345372)
[PASS] test_WrongRpId() (gas: 356177)
Suite result: ok. 15 passed; 0 failed; 0 skipped; finished in 498.50ms (65.05ms CPU time)

Ran 53 tests for test/WorldIDRegistry.t.sol:WorldIDRegistryTest
[PASS] test_CannotCreateAccountWithInsufficientFee() (gas: 6337766)
[PASS] test_CannotCreateManyAccountsWithInsufficientFee() (gas: 6343556)
[PASS] test_CannotRecoverAccountWhichHasNoRecoveryAgent() (gas: 645378)
[PASS] test_CannotRegisterAuthenticatorAddressThatIsAlreadyInUse() (gas: 624243)
[PASS] test_CannotSetFeeRecipientToZeroAddress() (gas: 6261603)
[PASS] test_CannotSetFeeTokenToZeroAddress() (gas: 6260433)
[PASS] test_CreateAccount() (gas: 617272)
[PASS] test_CreateAccountWithExcessFee() (gas: 6715734)
[PASS] test_CreateAccountWithFee() (gas: 6675522)
[PASS] test_CreateAccountWithNoRecoveryAgent() (gas: 665749)
[PASS] test_CreateManyAccounts() (gas: 6643006)
[PASS] test_CreateManyAccountsWithFee() (gas: 6826106)
[PASS] test_GetLatestRoot() (gas: 615253)
[PASS] test_GetMaxAuthenticators() (gas: 26562)
[PASS] test_GetNextLeafIndex() (gas: 614096)
[PASS] test_GetPackedAccountData() (gas: 617342)
[PASS] test_GetRecoveryAddress() (gas: 612067)
[PASS] test_GetRecoveryCounter() (gas: 613144)
[PASS] test_GetRootTimestamp() (gas: 24200)
[PASS] test_GetRootValidityWindow() (gas: 26031)
[PASS] test_GetSignatureNonce() (gas: 1201270)
[PASS] test_GetTreeDepth() (gas: 16360)
[PASS] test_InsertAuthenticatorDuplicatePubkeyId() (gas: 624323)
[PASS] test_InsertAuthenticatorDuplicatePubkeyIdNonZeroIndex() (gas: 648000)
[PASS] test_InsertAuthenticatorSuccess() (gas: 1223162)
[PASS] test_MockERC1271Wallet_Validation() (gas: 279557)
[PASS] test_OnlyOwnerCanSetFeeRecipient() (gas: 6263841)
[PASS] test_OnlyOwnerCanSetFeeToken() (gas: 6815947)
[PASS] test_OnlyOwnerCanSetRegistrationFee() (gas: 6241628)
[PASS] test_RecoverAccountSuccess() (gas: 1588599)
[PASS] test_RecoverAccountWithERC1271Wallet() (gas: 1504025)
[PASS] test_RemoveAuthenticatorSuccess() (gas: 1203146)
[PASS] test_SetFeeRecipient() (gas: 6266700)
[PASS] test_SetFeeToken() (gas: 6820307)
[PASS] test_SetMaxAuthenticators() (gas: 634824)
[PASS] test_SetMaxAuthenticators_RevertWhen_ValueAboveLimit() (gas: 31113)
[PASS] test_SetMaxAuthenticators_SucceedsAtMaxValidValue() (gas: 23673)
[PASS] test_SetMaxAuthenticators_SucceedsBelowMaxValue() (gas: 35441)
[PASS] test_SetRegistrationFee() (gas: 6265966)
[PASS] test_TreeDepth() (gas: 16160)
[PASS] test_UpdateAuthenticatorInvalidLeafIndex() (gas: 664273)
[PASS] test_UpdateAuthenticatorInvalidNonce() (gas: 681276)
[PASS] test_UpdateAuthenticatorSuccess() (gas: 1207610)
[PASS] test_UpdateRecoveryAddressToZeroAddress() (gas: 659882)
[PASS] test_UpdateRecoveryAddress_RevertInvalidNonce() (gas: 639147)
[PASS] test_UpdateRecoveryAddress_SetNewAddress() (gas: 663819)
[PASS] test_isValidRoot_customValidityWindow() (gas: 633077)
[PASS] test_isValidRoot_expiresAfterWindow() (gas: 630701)
[PASS] test_isValidRoot_latestRootAlwaysValid() (gas: 32265)
[PASS] test_isValidRoot_unknownRootReturnsFalse() (gas: 18455)
```

```
[PASS] test_isValidRoot_zeroWindowMakesHistoricalRootsInvalid() (gas: 631272)
[PASS] test_setRootValidityWindow_emitsEvent() (gas: 30352)
[PASS] test_setRootValidityWindow_onlyOwner() (gas: 17060)
Suite result: ok. 53 passed; 0 failed; 0 skipped; finished in 498.53ms (304.00ms CPU time)

Ran 4 tests for test/BinaryIMT.t.sol:BinaryIMTTest
[PASS] test_InsertManyCorrectness() (gas: 262900215)
[PASS] test_InsertManyTree() (gas: 9606320)
[PASS] test_InsertTree() (gas: 253294935)
[PASS] test_UpdateTree() (gas: 2099304)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 2.04s (4.14s CPU time)

Ran 10 test suites in 2.04s (4.10s CPU time): 184 tests passed, 0 failed, 0 skipped (184 total tests)
```

8.3 Automated Tools

8.3.1 AuditAgent

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.

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 conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to 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.

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.