Security Review Report
NM-0687 - JustanAccount

NETHERMIND SECURITY

(October 21, 2025)

Contents

1 Executive Summary

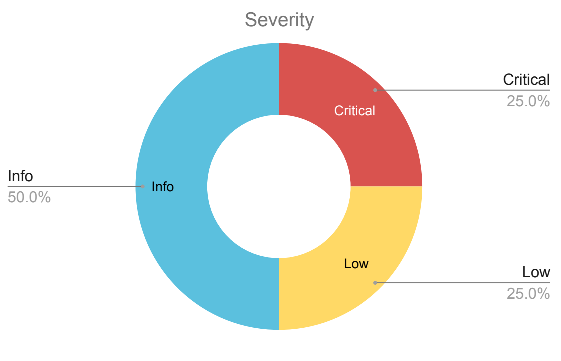
This document presents the results of a security review conducted by Nethermind Security for JustanAccount contract.

The `JustanAccount` is a Solidity smart contract designed to enhance Ethereum account functionalities by integrating support for EIP-7702 and EIP-4337. These integrations enable features such as transaction batching, gas fee sponsorship, cross-chain owner synchronization, and advanced signature validation.

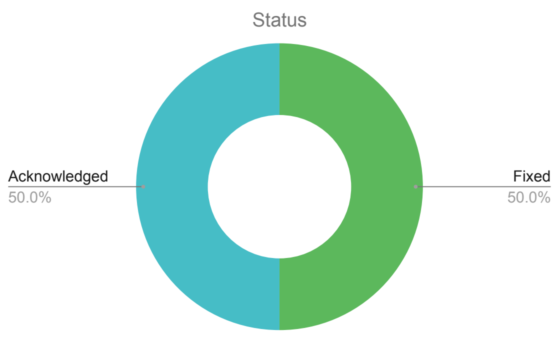
The audit comprises 371 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report four points of attention, where one is classified as `Critical`, one is classified as `Low`, and two are classified as `Informational`. 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 test suite evaluation and automated tools used. Section 9 concludes the document.



(a)



(b)

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

Summary of the Audit

Audit Type	Security Review
Initial Report	October 20, 2025
Final Report	October 21, 2025
Initial Commit	d6a7ed2483d53286159ca0b3ce0365f9c9f04d43
Final Commit	1c309a1663ddba8f6b4990e5450ebdf77b18380e
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/MultiOwnable.sol	115	176	153.0%	41	332
2	src/JustanAccountFactory.sol	41	47	114.6%	15	103
3	src/JustanAccount.sol	215	153	71.2%	45	413
	Total	**371**	**376**	**101.3%**	**101**	**848**

3 Summary of Issues

	Finding	Severity	Update
1	EIP-7702 delegation allows front-running attack on account initialization	Critical	Fixed
2	Gas griefing via WebAuthn signatures	Low	Acknowledged
3	Stuck nonce can break cross-chain replayability	Info	Acknowledged
4	`getUserOpHashWithoutChainId()` uses unhashed name and version for domain separator	Info	Fixed

4 Protocol Overview

The JustanAccount project is a smart contract account system designed to enhance the functionality of Ethereum accounts by integrating support for EIP-4337 (Account Abstraction) and EIP-7702 (delegated implementation for EOAs). The architecture is composed of three primary contracts: `JustanAccountFactory`, `JustanAccount`, and `MultiOwnable`.

The `JustanAccountFactory` contract serves as the deployment entry point. It utilizes the `CREATE2` opcode to deploy minimal, gas-efficient ERC-1967 proxies for each new account. This approach ensures that account addresses are deterministic and consistent across all supported EVM chains, calculated from a salt derived from the initial owners and a nonce. All proxy instances point to a single, shared `JustanAccount` implementation contract.

The core of the system is the `JustanAccount` contract, which functions as the smart wallet itself. It inherits from a `BaseAccount` (for ERC-4337 compliance), `MultiOwnable` (for owner management), and `ERC1271` (for signature validation). Its dual compliance with EIP-4337 and EIP-7702 allows it to operate either as a native smart contract account or as an extension that grants smart contract capabilities to an existing EOA.

Ownership and authorization are managed through the `MultiOwnable` logic. A key feature is its support for multiple owners of flexible types. Owners can be traditional 20-byte Ethereum addresses or 64-byte WebAuthn public keys (represented as x, y coordinates), enabling modern authentication methods.

The system offers several advanced features:

- **Transaction Batching:** The `executeBatch()` function allows multiple operations to be executed atomically in a single transaction, improving efficiency and user experience.
- **Cross-Chain Owner Synchronization:** The contract supports replaying owner management operations (e.g., adding or removing an owner) across multiple chains with a single signature. This is facilitated by the `executeWithoutChainIdValidation()` function, which bypasses the chain ID check for a predefined whitelist of secure functions.
- **Advanced Signature Validation:** The contract supports both standard ECDSA signatures and P-256 signatures for WebAuthn authentication.
- **Collision-Resistant Storage:** To ensure safe usage as a delegated implementation (EIP-7702), the contract utilizes ERC-7201 for namespaced storage, preventing potential storage layout collisions.

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 [Critical] EIP-7702 delegation allows front-running attack on account initialization

File(s): `src/JustanAccount.sol`

Description: The `JustanAccount` contract is designed as an implementation template for an Externally Owned Account (EOA) to use via **EIP-7702 delegation**. This mechanism allows the EOA to execute the contract's logic in its own context, including the `initialize(...)` function which is used to set the initial owners of the account.

The `initialize(...)` function has `external` visibility and is intended to be called only once when the account logic is first used to set the owners. The only restriction on who can call it is a check that the account is not already initialized (`nextOwnerIndex() == 0`).

```
function initialize(bytes[] calldata owners) external payable virtual {
    //
    if (nextOwnerIndex() != 0) {
        revert JustanAccount_AlreadyInitialized();
    }

    _initializeOwners(owners);
}
```

In an EIP-7702 flow, the EOA first signs a delegation tuple, which is then submitted to the network.

A malicious actor can observe the EOA's delegation tuple and/or the transaction initializing the contract in the mempool. If they submit a transaction to the EOA's address, calling the delegated `initialize(...)` function with the attacker's address as the owner, and this transaction is mined before the EOA's intended initialization, the attacker can successfully front-run the EOA.

Once the attacker's transaction is executed, they become the owner of the EOA's account logic in the EOA's context. The attacker can then call privileged functions, such as `execute(...)`, to drain all Ether and ERC-20 tokens held by the victim EOA. This allows an attacker to seize control and steal all funds from any EOA that attempts to use the `JustanAccount` for EIP-7702 delegation.

Recommendation(s): Consider restricting the caller of the `initialize(...)` function to ensure that only the EOA being initialized can set its own owners in case of use of EIP-7702, or the factory contract for ERC4337.

Status: Fixed.

Update from the client: We've successfully addressed the front-running vulnerability in the initialize() function. Our solution restructures the deployment architecture: the JustanAccountFactory now deploys the JustanAccount implementation in its constructor and passes its own address as an authorized factory parameter. The initialize() function is now restricted to only accept calls from the authorized factory address, completely preventing unauthorized initialization attempts. For EIP-7702 EOAs, users add owners via self-calls to addOwnerAddress() or addOwnerPublicKey(), which are protected by the existing ownership checks that permit msg.sender == address(this).

Fixed in commit 19b002.

6.2 [Low] Gas griefing via WebAuthn signatures

File(s): `src/JustanAccount.sol`

Description: The `JustanAccount` contract is an ERC-4337 compliant smart contract account that validates `UserOperations` through its `validateUserOp(...)` function. The signature validation logic supports multiple schemes, including WebAuthn for owners with registered public keys. When a signature does not match the format of a standard ECDSA signature, it is processed as a potential WebAuthn signature.

The validation path for WebAuthn signatures is susceptible to a gas griefing attack. The core of the issue lies in the use of the `WebAuthn.tryDecodeAuth(...)` function , which is used to parse the `WebAuthnAuth` struct from the signature data. This decoding function successfully parses the struct even if arbitrary data is appended to a valid encoded payload. It does not enforce that the entire input byte array is consumed during the decoding process.

This allows an attacker to take a valid WebAuthn signature for a `UserOperation` and append a large amount of arbitrary data to it. When this bloated signature is submitted to the ERC-4337 `EntryPoint`, the account's validation logic will succeed because `tryDecodeAuth(...)` correctly decodes the valid portion of the signature while ignoring the appended data. By increasing the size of the `signature` field, the attacker forces the account owner or their paymaster to incur higher gas costs. This can be used to drain the account's gas deposit.

Recommendation(s): Consider adding a validation step to ensure that the length of the `signatureData` corresponds to a reasonably encoded `WebAuthnAuth` struct without extraneous data.

Status: Acknowledged.

Update from the client: We acknowledge this gas griefing vulnerability where a malicious bundler could append arbitrary data to a valid WebAuthn signature. However, the practical impact is inherently limited by ERC-4337's verificationGasLimit field, which caps the maximum gas consumption during signature verification.

6.3 [Info] Stuck nonce can break cross-chain replayability

File(s): `src/JustanAccount.sol`

Description: The `JustanAccount` contract implements a cross-chain replayability feature through the `executeWithoutChainIdValidation(...)` function. This functionality allows owner management operations to be executed across multiple chains with a single signature. To manage the nonces for these special operations, a constant key, `REPLAYABLE_NONCE_KEY`, is used. The `validateUserOp(...)` function checks if a `UserOperation` targets `executeWithoutChainIdValidation(...)` and ensures that the `REPLAYABLE_NONCE_KEY` is used for its nonce.

The ERC-4337 `EntryPoint`, which acts as a nonce manager, enforces strict sequential ordering for nonces associated with the same key. This means that for the `REPLAYABLE_NONCE_KEY`, `UserOperation`s must be processed in the exact order of their sequence number (e.g., 0, 1, 2, ...).

The problem arises when a replayable `UserOperation` fails on one chain but succeeds on others. For instance, a user might broadcast a transaction to remove an owner across three different chains. If the account on one of those chains is in a state where the removal is invalid (e.g., the owner was never added on that chain, or it's the last owner), the `UserOperation` will revert. As a result, the `EntryPoint` will not increment the nonce for the `REPLAYABLE_NONCE_KEY` on that specific chain.

This creates a permanent desynchronization. All subsequent replayable operations with higher nonces will fail on the affected chain because the `EntryPoint` will reject them as being out of order. This effectively breaks the cross-chain synchronization feature for that account on that chain. This can be fixed by manually crafting and sending a separate transaction to call `incrementNonce(...)` on the `EntryPoint` contract.

Recommendation(s): Consider documenting this possibility and explaining the best practices to follow when doing owner management on `JustanAccount` contracts. Also document how to fix this scenario in case it is reached.

Status: Acknowledged.

6.4 [Info] `getUserOpHashWithoutChainId()` uses unhashed name and version for domain separator

File(s): `src/JustanAccount.sol`

Description: The `getUserOpHashWithoutChainId(...)` function is intended to compute a hash for a `UserOperation` that is valid across multiple chains by omitting the `chainId` from the EIP-712 domain separator . This allows for the replay of specific transactions, such as owner management, across different networks .

The function constructs the domain separator using the raw strings `"ERC4337"` and `"1"` for the `name` and `version` fields, respectively . This implementation deviates from the EIP-712 standard, which specifies that the `string` components of the `EIP712Domain` struct should be `keccak256` hashed before being included in the final domain separator hash. The `EIP712` contract inherited by `EntryPoint` correctly follows this standard by hashing the name and version when building its own domain separator .

This inconsistency means that any off-chain signer or wallet that correctly implements the EIP-712 specification will compute a different `userOpHash` than the one generated on-chain by `getUserOpHashWithoutChainId(...)`. As a result, signatures may fail validation when processed by the `JustanAccount` contract for cross-chain operations, undermining the intended functionality.

```

function getUserOpHashWithoutChainId(PackedUserOperation calldata userOp) public view virtual returns (bytes32) {
    bytes32 overrideInitCodeHash = Eip7702Support._getEip7702InitCodeHashOverride(userOp);
    // @audit-issue The name "ERC4337" and version "1" are passed as raw strings instead of their keccak256 hashes.
    return MessageHashUtils.toTypedDataHash(
        keccak256(abi.encode(TYPE_HASH, "ERC4337", "1", 0, address(entryPoint()))),
        userOp.hash(overrideInitCodeHash)
    );
}
```

Recommendation(s): Consider aligning the implementation of `getUserOpHashWithoutChainId(...)` with the EIP-712 standard by hashing the `name` and `version` strings before encoding them into the domain separator.

Status: Fixed.

Update from the client: The implementation has been corrected to properly comply with the EIP-712 standard by hashing the name and version string fields before encoding them into the domain separator.

Fixed in commit e258c2.

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 JustanAccount documentation

The JustaLab team provided a clear and comprehensive overview of the JustanAccount contract during the kick-off call. This was supported with written documentation detailing the different features present in the contracts.

8 Test Suite Evaluation

8.1 Tests Output

```
> forge test
[] Compiling...
[] Compiling 106 files with Solc 0.8.30
[] Solc 0.8.30 finished in 5.31s
Compiler run successful with warnings:
Warning (2072): Unused local variable.
  --> test/integration/Test4337ExecuteFlow.t.sol:42:9:
   |
42 |         bytes memory signature = abi.encodePacked(r, s, v);
   |         ^^^^^^^^^^^^^^^^^^^^^^

Warning (2072): Unused local variable.
  --> test/integration/Test7702ExecuteFlow.t.sol:32:9:
   |
32 |         bytes memory signature = abi.encodePacked(r, s, v);
   |         ^^^^^^^^^^^^^^^^^^^^^^

Warning (2018): Function state mutability can be restricted to view
   --> test/unit/Test7702SignatureValidation.t.sol:125:5:
    |
125 |     function test_ShouldValidateECDSAPersonalSign(string memory message) public {
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
   --> test/unit/Test7702SignatureValidation.t.sol:139:5:
    |
139 |     function test_ShouldValidateECDSAEIP712StructuredMessage() public {
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to pure
   --> test/unit/TestCrossChainValidation.t.sol:459:5:
    |
459 |     function _signWebAuthn(
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
   --> test/unit/TestWebAuthnValidation.t.sol:130:5:
    |
130 |     function test_ShouldValidateWebAuthnPersonalSign(string memory message) public {
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
   --> test/unit/TestWebAuthnValidation.t.sol:160:5:
    |
160 |     function test_ShouldValidateWebAuthnEIP712StructuredMessage() public {
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/unit/TestWrappedSignatureValidation.t.sol:54:5:
   |
54 |     function test_ShouldValidateECDSAPersonalSign(string memory message) public {
   |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/unit/TestWrappedSignatureValidation.t.sol:71:5:
   |
71 |     function test_ShouldValidateECDSAEIP712StructuredMessage() public {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Ran 9 tests for test/unit/TestJustanAccountFactory.t.sol:TestJustanAccountFactory
[PASS] test_DeployDeterministicPassValues() (gas: 266330)
[PASS] test_RevertsIfLength32ButLargerThanAddress() (gas: 296695)
[PASS] test_constructor_deploysImplementation() (gas: 3153421)
[PASS] test_createAccountDeploysToPredeterminedAddress() (gas: 270001)
[PASS] test_createAccountSetsOwnersCorrectly() (gas: 274844)
[PASS] test_implementation_returnsExpectedAddress() (gas: 10514)
[PASS] test_initCodeHash() (gas: 10681)
```

```
[PASS] test_revertsIfAccountAlreadyExists() (gas: 263398)
[PASS] test_revertsIfNoOwners() (gas: 29096)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 11.99ms (2.84ms CPU time)

Ran 17 tests for test/unit/TestCanSkipChainIdValidation.t.sol:TestCanSkipChainIdValidation
[PASS] test_ShouldReturnConsistentResultForSameSelector() (gas: 11396)
[PASS] test_ShouldReturnFalseForEntryPoint() (gas: 8655)
[PASS] test_ShouldReturnFalseForExecute() (gas: 8623)
[PASS] test_ShouldReturnFalseForExecuteBatch() (gas: 8631)
[PASS] test_ShouldReturnFalseForExecuteWithoutChainIdValidation() (gas: 8656)
[PASS] test_ShouldReturnFalseForInitialize() (gas: 8656)
[PASS] test_ShouldReturnFalseForIsValidSignature() (gas: 8678)
[PASS] test_ShouldReturnFalseForMaxSelector() (gas: 8655)
[PASS] test_ShouldReturnFalseForRandomSelector(bytes4) (runs: 256, : 10129, ~: 10129)
[PASS] test_ShouldReturnFalseForSupportsInterface() (gas: 8678)
[PASS] test_ShouldReturnFalseForValidateUserOp() (gas: 8676)
[PASS] test_ShouldReturnFalseForZeroSelector() (gas: 8632)
[PASS] test_ShouldReturnTrueForAddOwnerAddress() (gas: 8587)
[PASS] test_ShouldReturnTrueForAddOwnerPublicKey() (gas: 8562)
[PASS] test_ShouldReturnTrueForAllApprovedSelectors() (gas: 15099)
[PASS] test_ShouldReturnTrueForRemoveLastOwner() (gas: 8643)
[PASS] test_ShouldReturnTrueForRemoveOwnerAtIndex() (gas: 8649)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 17.26ms (8.16ms CPU time)

Ran 5 tests for test/unit/TestMultiOwnableContractSelf.t.sol:TestMultiOwnableContractSelf
[PASS] test_ShouldAllowBothContractSelfAndOwnerAccess(address,bytes32,bytes32) (runs: 256, : 179733, ~: 179673)
[PASS] test_ShouldAllowContractSelfToAddOwnerAddress(address) (runs: 256, : 105953, ~: 105953)
[PASS] test_ShouldAllowContractSelfToAddOwnerPublicKey(bytes32,bytes32) (runs: 256, : 128312, ~: 128390)
[PASS] test_ShouldAllowContractSelfToRemoveLastOwner(address) (runs: 256, : 102166, ~: 102152)
[PASS] test_ShouldAllowContractSelfToRemoveOwnerAtIndex(address,address) (runs: 256, : 161859, ~: 161827)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 60.42ms (59.34ms CPU time)

Ran 2 tests for test/integration/TestMultiOwnableFlow.t.sol:TestMultiOwnableFlow
[PASS] test_ShouldChangeOwnershipCorrectlyWith4337(address,address) (runs: 256, : 329982, ~: 329949)
[PASS] test_ShouldChangeOwnershipCorrectlyWith7702(address,address,address,bytes32,bytes32,bytes32,bytes32) (runs: 256,
↪ : 420613, ~: 420466)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 243.63ms (233.34ms CPU time)

Ran 27 tests for test/unit/TestMultiOwnableWithOwners.t.sol:TestMultiOwnableWithOwners
[PASS] test_AddOwnerAtNonContiguousIndex(address,address,address) (runs: 256, : 293661, ~: 293661)
[PASS] test_NextOwnerIndexAfterRemovals(address,address) (runs: 256, : 215346, ~: 215313)
[PASS] test_OwnerAtIndexAfterMultipleRemovals(address,address,address) (runs: 256, : 230960, ~: 230921)
[PASS] test_ShouldAddOwnerAddressCorrectly(address) (runs: 256, : 101269, ~: 101269)
[PASS] test_ShouldAddOwnerPublicKeyCorrectly(bytes32,bytes32) (runs: 256, : 123340, ~: 123340)
[PASS] test_ShouldRemoveLastOwnerCorrectly() (gas: 45439)
[PASS] test_ShouldRemoveOwnerAtIndexCorrectly(address) (runs: 256, : 123700, ~: 123700)
[PASS] test_ShouldReturnCorrectNextOwnerIndex() (gas: 11010)
[PASS] test_ShouldReturnCorrectOwnerAtIndex() (gas: 16928)
[PASS] test_ShouldReturnCorrectOwnerCount() (gas: 13197)
[PASS] test_ShouldReturnEmptyBytesForEmptyIndex() (gas: 12110)
[PASS] test_ShouldReturnFalseForNonOwnerAddress(address) (runs: 256, : 14015, ~: 14015)
[PASS] test_ShouldReturnFalseForNonOwnerBytes(address) (runs: 256, : 14549, ~: 14549)
[PASS] test_ShouldReturnFalseForNonOwnerPublicKey(bytes32,bytes32) (runs: 256, : 11488, ~: 11488)
[PASS] test_ShouldReturnTrueForOwnerAddress() (gas: 13497)
[PASS] test_ShouldReturnTrueForOwnerBytes() (gas: 14007)
[PASS] test_ShouldReturnTrueForOwnerPublicKey(bytes32,bytes32) (runs: 256, : 112525, ~: 112525)
[PASS] test_ShouldReturnZeroRemovedOwnersCount() (gas: 10967)
[PASS] test_ThrowErrorIfAddingDuplicateOwnerAddress() (gas: 20436)
[PASS] test_ThrowErrorIfAddingDuplicateOwnerPublicKey(bytes32,bytes32) (runs: 256, : 114635, ~: 114635)
[PASS] test_ThrowErrorIfNonOwnerAddsOwnerAddress(address) (runs: 256, : 17850, ~: 17850)
[PASS] test_ThrowErrorIfNonOwnerAddsOwnerPublicKey(address,bytes32,bytes32) (runs: 256, : 17868, ~: 17868)
[PASS] test_ThrowErrorIfNonOwnerRemovesOwner(address) (runs: 256, : 18539, ~: 18539)
[PASS] test_ThrowErrorIfRemoveLastOwnerWithMultipleOwners(address) (runs: 256, : 94696, ~: 94696)
[PASS] test_ThrowErrorIfRemovingLastOwner() (gas: 19089)
[PASS] test_ThrowErrorIfRemovingOwnerFromEmptyIndex(address) (runs: 256, : 97410, ~: 97410)
[PASS] test_ThrowErrorIfWrongOwnerAtIndex(address) (runs: 256, : 100867, ~: 100867)
Suite result: ok. 27 passed; 0 failed; 0 skipped; finished in 262.49ms (251.48ms CPU time)

Ran 1 test for test/integration/Test4337ExecuteFlow.t.sol:Test4337ExecuteFlow
[PASS] test_ShouldExecute4337FlowCorrectly(address,uint256,bytes32) (runs: 256, : 239365, ~: 240012)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 262.46ms (253.57ms CPU time)

Ran 1 test for test/integration/Test7702ExecuteFlow.t.sol:Test7702ExecuteFlow
[PASS] test_ShouldExecute7702FlowCorrectly(address,uint256,bytes32) (runs: 256, : 90886, ~: 91659)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 284.08ms (275.20ms CPU time)
```

```
Ran 20 tests for test/unit/TestExecuteWithoutChainIdValidation.t.sol:TestExecuteWithoutChainIdValidation
[PASS] test_ShouldHandleAddAndRemoveInSameBatch(address,address) (runs: 256, : 162448, ~: 162595)
[PASS] test_ShouldMaintainStateAcrossMultipleCalls(address,address,address) (runs: 256, : 258214, ~: 258914)
[PASS] test_ShouldPropagateRevertData() (gas: 28627)
[PASS] test_ShouldRevertWhenApprovedCallFails(uint256,address) (runs: 256, : 108532, ~: 108688)
[PASS] test_ShouldRevertWhenCallerNotOwnerOrEntryPoint(address,address) (runs: 256, : 23005, ~: 23005)
[PASS] test_ShouldRevertWhenOneCallHasDisallowedSelector(address) (runs: 256, : 108531, ~: 108609)
[PASS] test_ShouldRevertWithAlreadyOwnerError() (gas: 29896)
[PASS] test_ShouldRevertWithDisallowedSelector_Execute(address) (runs: 256, : 20355, ~: 20355)
[PASS] test_ShouldRevertWithDisallowedSelector_ExecuteBatch() (gas: 20315)
[PASS] test_ShouldRevertWithRandomDisallowedSelector(bytes4) (runs: 256, : 21582, ~: 21582)
[PASS] test_ShouldSucceedViaEIP7702Delegation(address) (runs: 256, : 110675, ~: 110831)
[PASS] test_ShouldSucceedWhenCalledByEntryPoint(address) (runs: 256, : 101715, ~: 101715)
[PASS] test_ShouldSucceedWhenCalledByOwner(address) (runs: 256, : 102123, ~: 102123)
[PASS] test_ShouldSucceedWithApprovedSelector_AddOwnerAddress(address) (runs: 256, : 105884, ~: 105884)
[PASS] test_ShouldSucceedWithApprovedSelector_AddOwnerPublicKey(bytes32,bytes32) (runs: 256, : 127831, ~: 127831)
[PASS] test_ShouldSucceedWithApprovedSelector_RemoveLastOwner() (gas: 54306)
[PASS] test_ShouldSucceedWithApprovedSelector_RemoveOwnerAtIndex(address) (runs: 256, : 131211, ~: 131367)
[PASS] test_ShouldSucceedWithEmptyCallsArray() (gas: 24054)
[PASS] test_ShouldSucceedWithManyApprovedCalls() (gas: 791439)
[PASS] test_ShouldSucceedWithMultipleApprovedSelectors(address,address,bytes32,bytes32) (runs: 256, : 276126, ~: 276437)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 296.02ms (285.74ms CPU time)

Ran 6 tests for test/unit/TestWrappedSignatureValidation.t.sol:TestWrappedSignatureValidation
[PASS] test_ShouldFailWrappedSignatureWithRemovedOwner(uint256) (runs: 256, : 131995, ~: 132005)
[PASS] test_ShouldFailWrappedSignatureWithWrongSigner(uint256,uint256) (runs: 256, : 144031, ~: 144031)
[PASS] test_ShouldFailWrappedSignatureWithoutOwner(uint256) (runs: 256, : 62022, ~: 62022)
[PASS] test_ShouldValidateECDSAEIP712StructuredMessage() (gas: 29874)
[PASS] test_ShouldValidateECDSAPersonalSign(string) (runs: 256, : 30449, ~: 30452)
[PASS] test_ShouldValidateWrappedSignature(uint256) (runs: 256, : 140782, ~: 140782)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 358.24ms (391.26ms CPU time)

Ran 6 tests for test/unit/Test7702JustanAccount.t.sol:Test7702JustanAccount
[PASS] test_ShouldExecuteBatchCorrectly(address,uint256) (runs: 256, : 52870, ~: 53116)
[PASS] test_ShouldExecuteCorrectly(address,uint256) (runs: 256, : 65871, ~: 66027)
[PASS] test_ShouldThrowErrorIfSponsoringExecute(uint256) (runs: 256, : 16911, ~: 16911)
[PASS] test_ShouldThrowErrorIfSponsoringExecuteBatch(address,uint256) (runs: 256, : 18721, ~: 18721)
[PASS] test_ThrowErrorIfCallingExecuteBatchFromNotEntrypointOrOwner(address,uint256,bytes) (runs: 256, : 13086, ~:
 ↪ 13080)
[PASS] test_ThrowErrorIfCallingExecuteFromNotEntrypointOrOwner(address,uint256,bytes) (runs: 256, : 12009, ~: 12007)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 438.83ms (508.84ms CPU time)

Ran 6 tests for test/unit/Test7702SignatureValidation.t.sol:Test7702SignatureValidation
[PASS] test_ShouldFailWhenNotDelegated(uint256) (runs: 256, : 55354, ~: 55354)
[PASS] test_ShouldFailWithOversizedSignature(uint256) (runs: 256, : 51296, ~: 51292)
[PASS] test_ShouldRevertWithUndersizedSignature(uint256) (runs: 256, : 47787, ~: 47790)
[PASS] test_ShouldValidateCorrectSignature() (gas: 51008)
[PASS] test_ShouldValidateECDSAEIP712StructuredMessage() (gas: 19654)
[PASS] test_ShouldValidateECDSAPersonalSign(string) (runs: 256, : 20175, ~: 20171)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 181.37ms (410.45ms CPU time)

Ran 4 tests for test/unit/Test4337JustanAccount.t.sol:Test4337JustanAccount
[PASS] test_ShouldExecuteBatchCallsCorrectly(address,uint256) (runs: 256, : 153482, ~: 153601)
[PASS] test_ShouldExecuteCallCorrectly(address,uint256) (runs: 256, : 170607, ~: 170918)
[PASS] test_ShouldValidateUserOpCorrectly(bytes) (runs: 256, : 39096, ~: 39060)
[PASS] test_ThrowErrorIfCallingValidateUserOpFromNotEntrypoint(address,bytes,uint256) (runs: 256, : 35107, ~: 35076)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 488.36ms (555.45ms CPU time)

Ran 11 tests for test/unit/TestGeneralJustanAccount.t.sol:TestGeneralJustanAccount
[PASS] test_ShouldAcceptValidEthereumAddressOwner(uint160) (runs: 256, : 178911, ~: 178911)
[PASS] test_ShouldFailInitializeWhenAlreadyInitialized() (gas: 179904)
[PASS] test_ShouldInitializeCorrectly() (gas: 265824)
[PASS] test_ShouldReceiveERC1155Correctly(uint256,uint256) (runs: 256, : 78035, ~: 78438)
[PASS] test_ShouldReceiveERC721Correctly(uint256) (runs: 256, : 113585, ~: 113600)
[PASS] test_ShouldReceiveEtherCorrectly(address,uint256) (runs: 256, : 17337, ~: 17495)
[PASS] test_ShouldReturnCorrectEntryPoint() (gas: 10583)
[PASS] test_ShouldReturnFalseIfIncorrectInterface(bytes4) (runs: 256, : 10469, ~: 10469)
[PASS] test_ShouldReturnTrueIfCorrectInterface() (gas: 14009)
[PASS] test_ShouldRevertOnInvalidEthereumAddressOwner(uint256) (runs: 256, : 88326, ~: 88326)
[PASS] test_ShouldRevertOnInvalidOwnerBytesLength(uint8) (runs: 256, : 93421, ~: 90670)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 587.35ms (299.49ms CPU time)

Ran 10 tests for test/unit/TestWebAuthnValidation.t.sol:TestWebAuthnValidation
[PASS] test_ShouldFailWebAuthnSignatureWithInvalidOwnerIndex(uint256) (runs: 256, : 70244, ~: 70244)
```

```
[PASS] test_ShouldFailWebAuthnSignatureWithRemovedOwner(uint256) (runs: 256, : 153822, ~: 153834)
[PASS] test_ShouldFailWebAuthnSignatureWithTamperedAuthenticatorData() (gas: 251176)
[PASS] test_ShouldFailWebAuthnSignatureWithTamperedClientDataJSON() (gas: 81647)
[PASS] test_ShouldFailWebAuthnSignatureWithWrongChallengeIndex() (gas: 81521)
[PASS] test_ShouldFailWebAuthnSignatureWithWrongKey(uint256,uint256) (runs: 256, : 344039, ~: 344040)
[PASS] test_ShouldFailWebAuthnSignatureWithWrongTypeIndex() (gas: 81567)
[PASS] test_ShouldValidateWebAuthnEIP712StructuredMessage() (gas: 223709)
[PASS] test_ShouldValidateWebAuthnPersonalSign(string) (runs: 256, : 222456, ~: 222439)
[PASS] test_ShouldValidateWebAuthnSignature() (gas: 252728)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 800.04ms (930.15ms CPU time)

Ran 14 tests for test/unit/TestCrossChainValidation.t.sol:TestCrossChainValidation
[PASS] test_ShouldComputeDifferentHashThanRegularUserOpHash() (gas: 27553)
[PASS] test_ShouldComputeHashWithoutChainId() (gas: 24055)
[PASS] test_ShouldFailWithTamperedCalldata(address,address) (runs: 256, : 43973, ~: 43973)
[PASS] test_ShouldFailWithWrongSignature(address) (runs: 256, : 36978, ~: 36978)
[PASS] test_ShouldProduceDifferentHashesForDifferentCallData(address,address) (runs: 256, : 26831, ~: 26831)
[PASS] test_ShouldProduceSameHashOnDifferentChains() (gas: 29392)
[PASS] test_ShouldRevertWhenRegularExecuteUsesReplayableNonceKey(address) (runs: 256, : 29672, ~: 29672)
[PASS] test_ShouldRevertWhenValidateUserOpNotCalledByEntryPointFuzzed(address,address) (runs: 256, : 29603, ~: 29603)
[PASS] test_ShouldRevertWithNonReplayableNonceKeyForCrossChain(uint256,address) (runs: 256, : 33594, ~: 33594)
[PASS] test_ShouldSucceedWhenExecuteWithoutChainIdValidationUsesReplayableNonceKey(address) (runs: 256, : 37724, ~:
 ↪ 37724)
[PASS] test_ShouldSucceedWhenRegularExecuteUsesNonReplayableNonceKey(uint256,address) (runs: 256, : 37034, ~: 37034)
[PASS] test_ShouldValidateCrossChainSignatureInEIP7702Mode(address) (runs: 256, : 43393, ~: 43393)
[PASS] test_ShouldValidateCrossChainWebAuthnSignature(address) (runs: 256, : 593693, ~: 594033)
[PASS] test_ShouldValidateSameSignatureAcrossChains(address) (runs: 256, : 71189, ~: 71189)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 782.57ms (1.10s CPU time)

Ran 15 test suites in 805.21ms (5.08s CPU time): 139 tests passed, 0 failed, 0 skipped (139 total tests)
```

8.2 Automated Tools

8.2.1 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.

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