# Security Review Report
# NM-0373 Summon

NETHERMIND
SECURITY

(Feb 19, 2025)

# Contents

# 1 Executive Summary

This document outlines the security review conducted by Nethermind Security for Summon. The audit focused on two separate smart contracts: `PaymentRouterNative` and `RewardsNative`. While both contracts are components of the Summon ecosystem, they serve distinct functions within different parts of the system and do not directly interact with each other. The `PaymentRouterNative` contract facilitates managing payments for in-game collectibles with configurable prices and URIs. The `RewardsNative` contract manages the distribution of rewards within the Summon ecosystem. It allows for the allocation of native token rewards to users who own the eligible access tokens.

**The audited code comprises** 603 lines of code written in the Solidity language. The audit focuses on the implementation of the Summon's `PaymentRouterNative` and `RewardsNative` contracts.

**The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report** nine points of attention, all of which are classified as `Informational` or `Best Practice`. 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. Section 9 concludes the document.

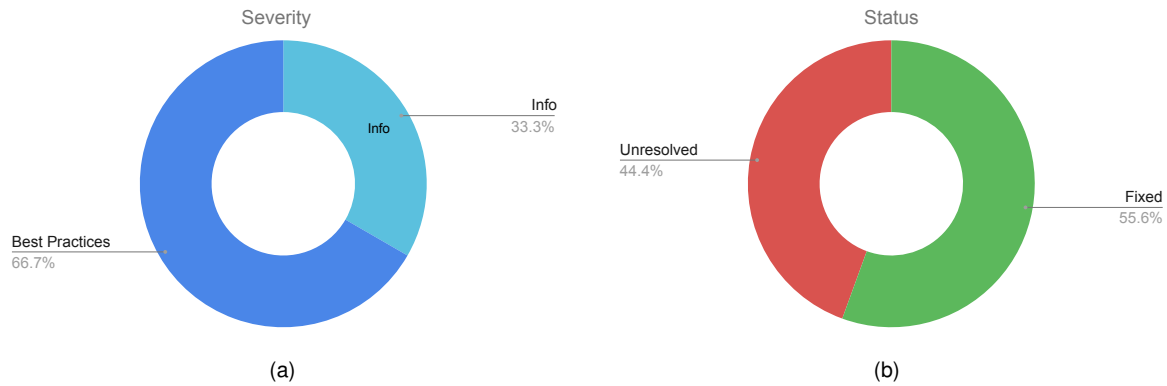


**Fig. 1: Distribution of issues: Critical** (0), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (3), **Best Practices** (6). **Distribution of status: Fixed** (5), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (4)

### Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | January 20, 2025 |
| **Final Report** | February 19, 2025 |
| **Repository** | contracts-monorepo |
| **Commit** | 0b85619c3df9849e67c4bfcd334500a98f4cdba9 |
| **Final Commit** | e60a44fea1de830da5ea50584b35ace59d30dfe6 |
| **Documentation** | PR description |
| **Documentation Assessment** | Low |
| **Test Suite Assessment** | Low |

## 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | contracts/payments/PaymentRouterNative.sol | 175 | 110 | 62.9% | 30 | 315 |
| 2 | contracts/soulbounds/RewardsNative.sol | 428 | 46 | 10.7% | 74 | 548 |
| | **Total** | **603** | **156** | **25.9%** | **104** | **863** |

## 3 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Signature checking from `ERCWhitelistSignature` is not used | Info | Fixed |
| 2 | The `RewardsNative` contract is supposed to handle native tokens only but inherits `ERC1155Holder` | Info | Unresolved |
| 3 | The `pay(...)` function might emit an incorrect sender address in the `PaymentReceived` event | Info | Fixed |
| 4 | Access control on `decodeData(...)` not required | Best Practices | Unresolved |
| 5 | Duplicated check for `_devWallet` address in the constructor | Best Practices | Fixed |
| 6 | Incorrect naming convention | Best Practices | Unresolved |
| 7 | Missing `address(0)` checks in the `constructor` | Best Practices | Unresolved |
| 8 | The `paymentURI(...)` function can re-use an existing modifier `validId(...)` to perform its checks | Best Practices | Fixed |
| 9 | Unused imports | Best Practices | Fixed |

# 4 System Overview

## 4.1 Actors

The `PaymentRouterNative` and `RewardsNative` contracts are distinct and don't interact directly. The set of actors and roles, however, is similar. The participants of both systems are outlined below.

- **Admin**: Address with the admin role can grant and revoke other roles. Admin in the `PaymentRouter` contract can update the Multisig address and withdraw the stuck funds from the contract.
- **Dev Config**: Address with Dev Config role can grant and revoke the whitelisted signer role. In the `PaymentRouterNative`, this role can also set and update Payment Config for specific payment IDs.
- **Manager**: Address with Manager role can pause and unpause the contracts. They can also create new reward tokens in the `RewardsNative` contract but must provide native token backing.
- **Minter**: Address with minter role can create new reward and access tokens and mint them to any address in the `RewardsNative` contract.
- **Whitelisted Signer**: Whitelisted Sigher address is operated by the project's backend and signs the data for collectibles that users can later pay for.
- **User**: Users pay for in-game collectibles or get access tokens, which can be later claimed for rewards.

## 4.2 Payment Router Native

The `PaymentRouterNative` contract manages payments for in-game collectibles within the Summon ecosystem. It allows for the configuration of payment IDs, each with a configurable price and associated metadata URI. The contract supports role-based access control, enabling different levels of permissions for managing payment configurations and performing critical operations, such as updating prices and URIs, pausing/unpausing payment IDs, and handling the multisig wallet address. Authorized backend services sign messages to verify that a user is allowed to purchase specific in-game collectibles. Users can then call the `pay(...)` function to complete the transaction using the Game7 native tokens. Payments are securely routed to a multisig wallet. This ensures that only authorized actions, validated by the signed messages, are executed within the system. The contract includes emergency withdrawal functionality to rescue any stuck funds from the contract.

## 4.3 Rewards Native

The `RewardsNative` contract facilitates the issuance and management of access tokens, which can be later redeemed for native token rewards. The contract includes functionalities for token creation, reward minting, and claiming rewards. The system allows stakeholders to mint reward tokens, subject to certain conditions and roles. The reward token can be created and minted without native token backing by the Minter. In this scenario, the tokens intended for distribution will be sent directly to the contract at a later date by other project stakeholders. Once that's done, users holding the `AccessToken` will be able to claim the rewards.

# 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 [Info] Signature checking from `ERCWhitelistSignature` is not used

**File(s)**: `contracts/soulbounds/RewardsNative.sol`

**Description**: The `RewardsNative` contract inherits from the `ERCWhitelistSignature` contract. While in other parts of the Summon ecosystem, this contract controls the user's actions, e.g., only allowing specific token IDs to be minted, this functionality remains unused in the `RewardsNative` contract.

**Recommendation(s)**: If `ERCWhitelistSignature` is not intended to be used, consider removing the inheritance specifier along with the unused functionality such as `devVerifySignature(...)`, `addWhitelistSigner(...)` and `removeWhitelistSigner(...)` functions.

**Status**: Fixed

**Update from the client**: 9f9a1ca

## 6.2 [Info] The `RewardsNative` contract is supposed to handle native tokens only but inherits `ERC1155Holder`

**File(s)**: `contracts/soulbounds/RewardsNative.sol`

**Description**: The `RewardsNative` contract is a simplified version of the older `Rewards` contract. Unlike its predecessor, it is supposed to handle only native token rewards. However, the `RewardsNative` contract still inherits the `ERC1155Holder` contract even though it has no functionality to manage this type of token.

**Recommendation(s)**: Consider removing the `ERC1155Holder` inheritance specifier and its override in the `supportsInterface(...)` function.

**Status**: Unresolved

## 6.3 [Info] The `pay(...)` function might emit an incorrect sender address in the `PaymentReceived` event

**File(s)**: `contracts/payments/PaymentRouterNative.sol`

**Description**: The `pay(...)` function uses the `_msgSender(...)` function from OpenZeppelin's `Context` library to perform the signature checks. Currently, this function returns the `msg.sender` address, which is the address of the user that called the `pay(...)` function. Using `_msgSender(...)` makes it possible to use proxies/paymasters that call the `pay(...)` function on behalf of the user in the future.

The `PaymentReceived` event, however, is not future-proofed. It emits the `msg.sender` address directly instead of using the address returned by the `_msgSender(...)` function. If paymaster functionality is added in the future, the emitted `sender` address won't point to the user but to the paymaster instead.

**Recommendation(s)**: Consider emitting the address returned by the `_msgSender(...)` function as the payment's `sender`.

**Status**: Fixed

**Update from the client**: eeddb97

## 6.4 [Best Practices] Access control on `decodeData(...)` not required

**File(s)**: `contracts/soulbounds/RewardsNative.sol`

**Description**: The `decodeData(...)` function can only be called by address with `DEV_CONFIG_ROLE`. But since it's a view function, there's no need to have access control.

**Recommendation(s)**: Consider removing the `onlyRole` modifier from `decodeData(...)`.

**Status**: Unresolved

## 6.5 [Best Practices] Duplicated check for `_devWallet` address in the constructor

**File(s)**: `contracts/soulbounds/RewardsNative.sol`

**Description**: The zero address check for the `_devWallet` address is performed twice in the constructor of the `RewardsNative` contract.

**Recommendation(s)**: Consider removing the redundant check.

**Status**: Fixed

**Update from the client**: 9f9a1ca

## 6.6 [Best Practices] Incorrect naming convention

**File(s)**: contracts/soulbounds/RewardsNative.sol

**Description**: The `_dangerous_createTokenAndDepositRewards` and `_dangerous_createMultipleTokensAndDepositRewards` functions are `public` and `external`, respectively, but their names suggest that it's an internal or private function.

**Recommendation(s)**: Consider updating the function name per solidity naming convention. Also, the `_dangerous_createTokenAndDepositRewards` function can be updated to `external`.

**Status**: Unresolved

**Update from the client**: 9f9a1ca

## 6.7 [Best Practices] Missing `address(0)` checks in the `constructor`

**File(s)**: contracts/payments/PaymentRouterNative.sol, contracts/soulbounds/RewardsNative.sol

**Description**: The constructor sets key parameters of the `PaymentRouterNative` contract, including the `managerRole` and `adminRole`'s addresses. The `_grantRole` function, however, does not check for the `address(0)`, which may lead to inadvertently granting one of the roles to the zero address. While the `_multiSigWallet` address is explicitly checked to ensure it is not zero, the `managerRole` and `adminRole` addresses lack this validation.

The `RewardsNative` contract's constructor accepts a set of address parameters and validates them not to be zero addresses. Such a check is not performed on the `_adminWallet` address.

**Recommendation(s)**: Consider checking that the provided `managerRole` and `adminRole` addresses are not zero. The same check should be added for the `_adminWallet` address in the `RewardsNative` contract.

**Status**: Unresolved

**Update from the client**: 9f9a1ca

## 6.8 [Best Practices] The `paymentURI(...)` function can re-use an existing modifier `validId(...)` to perform its checks

**File(s)**: contracts/payments/PaymentRouterNative.sol

**Description**: The `paymentURI(...)` function can retrieve the payment's URI given the payment ID. When an invalid ID is provided, the function will revert. The check is performed by ensuring that the price for that particular ID was configured and is non-zero.

```
function paymentURI(uint256 id) external view returns (string memory) {
    // @audit-issue The same check is performed by `validId(...)` modifier
    if (paymentConfigs[id].price == 0) revert InvalidPaymentId();
    return paymentConfigs[id].paymentURI;
}
```

The exact same check is already performed by the `validId(...)` modifier and could be re-used in the `paymentURI(...)` function to remove code duplication.

**Recommendation(s)**: Consider simplifying the logic inside the `paymentURI(...)` function by re-using an existing modifier to perform the input validation checks.

**Status**: Fixed

**Update from the client**: 9f9a1ca

## 6.9 [Best Practices] Unused imports

**File(s)**: contracts/soulbounds/RewardsNative.sol

**Description**: The `RewardsNative` contract imports utilities from OpenZeppelin, such as `ECDSA` and `Strings`. These imports, however, remain unused in the contract's logic.

**Recommendation(s)**: Consider removing unused imports.

**Status**: Fixed

**Update from the client**: 9f9a1ca

# 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 on Summon's Documentation**

The documentation for Summon's smart contracts was provided via the PR description. While the Summon team offered the Nethermind Security auditors the necessary background context to comprehend the contracts' functionality, readers lacking this context may find the code significantly harder to understand. To improve accessibility and clarity, it is strongly recommended that NatSpec comments be included in the contracts to explain their purpose and functionality.

# 8 Test Suite Evaluation

The Summon's test suite contains many test cases. The in-scope `PaymentRouterNative` and `RewardsNative` contracts could benefit from higher test coverage to outline the contract's specifications clearly. To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the results of the analysis performed on the Summon's smart contracts. The first column lists the contracts that were tested using mutation testing. The second column indicates the number of mutants generated and how many were "slain" (i.e., caught by the test suite). The third column provides the percentage of mutants slain, reflecting the effectiveness of the test suite in covering the particular contract. The higher the score, the better the test suite is at finding bugs.

| Contract | Mutants (slain / total generated) | Score |
|---|---|---|
| `PaymentRouterNative.sol` | 28 / 40 | 70% |
| `RewardsNative.sol` | 76 / 141 | 53.90% |
| **Total** | 104 / 181 | 57.46% |

The following is a list of code modifications not caught by the existing test suite. Each point highlights a scenario that could benefit from higher test coverage to enhance the protocol's overall security and resilience in the next code change iterations as the project evolves.

**PaymentRouterNative.sol**

- No test case ensures that the URI was properly set in the `setPaymentConfig(...)` function.
- No test case for a revert with `InvalidPaymentId` in the `paymentURI(...)` function.
- No test case ensures that an ID was unpaused successfully as a result of calling the `unpauseId(...)` function.
- No test case for a revert with `InvalidMultiSigAddress` in the `updateMultiSig(...)` function.
- No test case for a revert with `TransferToMultiSigFailed` in the `pay(...)` function.
- No test case ensures that the `_signer` was whitelisted correctly in the `addWhitelistSigner(...)` function. The logic inside this function can be removed entirely with no effect on the test suite.
- No test case ensures that the `_signer` was removed from the whitelist correctly in the `removeWhitelistSigner(...)` function. The logic inside this function can be removed entirely with no effect on the test suite.
- No test case for a revert with `InvalidSeed` in the `_verifyContractChainIdAndDecode(...)` function.
- No test case for a revert with `NoFundsToWithdraw` in the `withdrawStuckFunds(...)` function.
- No test case for a revert with `EmergencyWithdrawalFailed` in the `withdrawStuckFunds(...)` function.
- No test case ensures that the `adminRole` address was given the `DEFAULT_ADMIN_ROLE` after the contract's construction.

**RewardsNative.sol**

- No test case for a revert with `AddressIsZero` in the `updateRewardTokenContract(...)` function.
- No test case for a revert with `InsufficientBalance` in the `_transferEther(...)` function.
- No test case for a revert with `TransferFailed` in the `_transferEther(...)` function.
- No test case for a revert with `AddressIsZero` in the `_claimReward(...)` function.
- No test case ensures that `rewardTokenContract` was set properly as a result of calling the `updateRewardTokenContract(...)` function.
- No test case ensures that the reward token was burned as a result of calling the `_claimReward(...)` function.
- No test case for a revert with `AddressIsZero` in the `_mintRewardAccessToken(...)` function.
- No test case for a revert with `TokenNotExist` in the `_mintRewardAccessToken(...)` function.

- No test case for a revert with `InvalidAmount` in the `_mintRewardAccessToken(...)` function.
- No test case ensures that `currentRewardSupply[_tokenId]` was increased properly as a result of calling the `_mintRewardAccessToken(...)` function.
- The logic from the functions `_dangerous_createTokenAndDepositRewards(...)` and `_dangerous_createMultipleTokensAndDepositRewards(...)` can be removed entirely from the codebase with no effect on the test suite.
- No test case for a revert with `InvalidInput` in the `_verifyContractChainIdAndDecode(...)` function.
- The calls to `_createTokenAndDepositRewards(...)` and `_mintRewardAccessToken(...)` can be removed from the `createTokenAndMintRewards(...)` function with no effect on the test suite. The same is true for the batch versions of this function.
- No test case for a revert with `InsufficientBalance` in the `createTokenAndDepositRewards(...)` function.
- No test case asserts the amount of native tokens that must be provided during the call to `createMultipleTokensAndDepositRewards(...)`.
- No test case asserts the results of calling `updateTokenMintPaused(...)` and `updateClaimRewardPaused(...)`.
- No test case for a revert with `AddressIsZero` in the `withdrawAll(...)` function.
- No test case ensures that the rewards were claimed as a result of calling the `claimRewards(...)` function. The same is true for the batch version of the function.
- No test case for a revert with `InvalidLength` in the `batchMintById(...)` function.
- The logic from the functions `addWhitelistSigner(...)` and `removeWhitelistSigner(...)` can be removed with no effect on the test suite.
- No test case for a revert with `InvalidAmount` in the `_validateTokenInputs(...)` function.

## 8.1 Automated Tools

### 8.1.1 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 is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

**Smart Contract Development & DeFi Research:** Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

**Our suite of L2 tooling:** Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

**Learn more about us at nethermind.io**.

### General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

### Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.