Security Review Report
NM-0446 Obol

NETHERMIND
SECURITY

(Sep 05, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for Obol Network's Ethereum validator manager smart contracts. The `ObolValidatorManager` contract is responsible for managing Ethereum validators and distributing payments between principal and reward recipients. Additionally, features such as withdrawal requests and validators consolidation have been implemented to ensure compatibility with the Ethereum Pectra upgrade.

The audit comprises 287 lines of code written in Solidity language. It focused on the **ObolValidatorManagerFactory** and **ObolValidatorManager** smart 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** 7 points of attention, where one is classified as `Medium`, one is classified as `Low`, and five are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

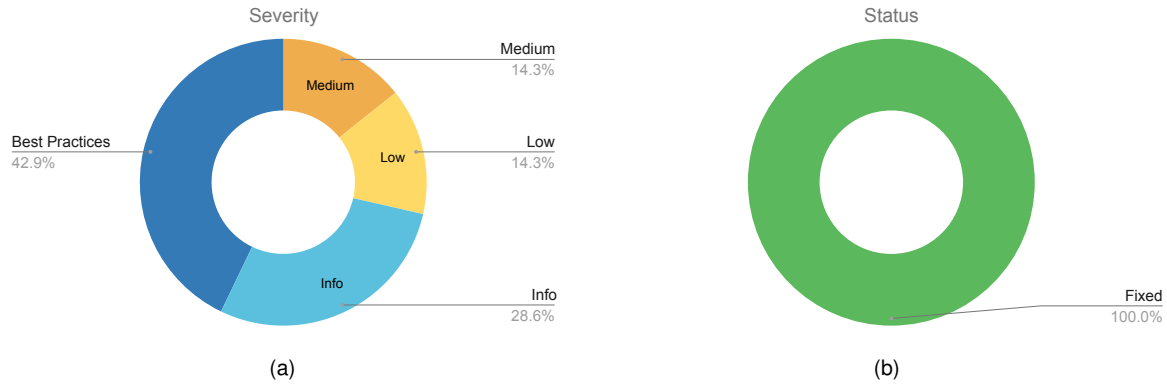

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (2), **Best Practices** (3). **Distribution of status: Fixed** (7), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 14, 2025
Response from Client	Regular responses during audit engagement
Final Report	September 05, 2025
Repository	obol-splits
Initial Commit	913232d
Final Commit	5e26749
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/ovm/ObolValidatorManagerFactory.sol	62	57	91%	17	136
2	src/ovm/ObolValidatorManager.sol	207	171	82%	85	461
3	src/interfaces/IDepositContract.sol	18	24	133.3%	6	48
	Total	**287**	**252**	**87.8%**	**108**	**645**

3 Summary of Issues

	Finding	Severity	Status
1	Deposits to Arbitrary Validators can Inflate Internal Accounting	Medium	Fixed
2	Possible reentrancy leads to a manipulated `fundsPendingWithdrawal` amount	Low	Fixed
3	Arbitrary external calls in the `recoverFunds` function	Info	Fixed
4	Call to arbitrary address in `withdraw`	Info	Fixed
5	Usage of hardcoded values	Best Practices	Fixed
6	Functions are marked payable, but don't use `msg.value`	Best Practices	Fixed
7	Lack of input sanitization for the contract `owner`	Best Practices	Fixed

4 System Overview

Obol built a set of smart contracts designed to manage Ethereum validators and distribute staking rewards and withdrawals between principal and reward recipients. The protocol consists of two main contracts:

- **ObolValidatorManagerFactory**: This contract acts as a factory, enabling the deployment of multiple `ObolValidatorManager` instances through the `createObolValidatorManager` function.
- **ObolValidatorManager**: This contract serves as the withdrawal address for validators. It is responsible for implementing the logic that handles the distribution of staking rewards and the management of validator withdrawals.

4.1 Payment Distribution Logic

The payment distribution mechanism in the `ObolValidatorManager` is designed to manage the inflow of funds to the contract. When the Ethereum network rewards the validator or the user initiates a partial/full withdrawal, the funds are deposited into the contract's balance.

The contract then classifies the funds into two categories: principal or reward amounts. It introduces a `principalThreshold` to help classify funds since tracking the exact source of the funds is not possible. Additionally, a wrapper deposit function is implemented to keep track of the total deposited amount. The distribution is then done as follows:

- Any funds below the threshold are classified as rewards and thus transferred to the Reward address.
- Any funds above the threshold, the tracked deposited amount is taken as principal and thus transferred to the principal address. While the extra amount is classified as a reward.

The `ObolValidatorManager` supports two modes for withdrawals: Push and Pull.

In the Push mode, the `distributeFunds` function is used to transfer Ether to the specified destination address directly.

```
function distributeFunds() external payable
```

In the Pull mode, the amount is stored in a `fundsPendingWithdrawal` variable. The funds can later be claimed by the user via a separate `withdraw` function.

```
function distributeFundsPull() external payable
```

4.2 Support for EIP-7251 and EIP-7002

To comply with the Pectra upgrade, the validator's withdrawal address must support functionalities for requesting withdrawals and consolidating validators. Therefore, two functions have been added to this contract to support these functionalities.

4.2.1 EIP-7251: Maximum Effective Balance Increase

The **EIP7251** increases the validator effective cap from 32 ETH to 2048 ETH, effectively allowing validators to stake any amount within the 32ETH to 2048ETH range. This reduces the need for multiple validator instances and introduces the ability of large validators to run fewer validators by consolidating balances from multiple validators into a single validator.

To support this functionality, the `requestConsolidation` function was introduced to the contract. This function initiates a consolidation request with the EIP-7251 system contract, where `source` validators are consolidated into a `target` validator. The function can only be called by actors with the `CONSOLIDATION_ROLE`.

```
function requestConsolidation(
    bytes[] calldata sourcePubKeys,
    bytes calldata targetPubKey
) external payable onlyOwnerOrRoles(CONSOLIDATION_ROLE)
```

4.2.2 EIP-7002: Execution Layer triggerable exits

The **EIP7002** allow validators to trigger exits and partial withdrawals via their execution layer (0x01) withdrawal credentials.

To support this functionality, the `requestWithdrawal` function was introduced. This function is exclusively callable by actors with the `WITHDRAWAL_ROLE` and serves to request a withdrawal from the EIP-7002 system contract.

```
function requestWithdrawal(
    bytes[] calldata pubKeys,
    uint64[] calldata amounts
) external payable onlyOwnerOrRoles(WITHDRAWAL_ROLE)
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Deposits to Arbitrary Validators can Inflate Internal Accounting

File(s): ObolValidatorManager.sol

Description: The `deposit` function allows any external caller to specify arbitrary `pubkey` and `withdrawal_credentials` when making a deposit into the beacon chain. However, the contract does not guarantee that the provided `withdrawal_credentials` correspond to this contract.

As a result, a malicious user can deposit to a validator with different withdrawal credentials, while the contract still records the deposit as principal by increasing `amountOfPrincipalStake`. This artificially inflates the internal accounting and creates the illusion that more ETH is staked than is actually the case.

Consequently, when the contract later distributes funds, excess ETH will be classified as principal even though it represents validator rewards. This can lead to incorrect accounting and misallocation of rewards.

```
function deposit(
    bytes calldata pubkey,
    bytes calldata withdrawal_credentials,
    bytes calldata signature,
    bytes32 deposit_data_root
  ) external payable {
    uint256 oldAmountOfPrincipalStake = amountOfPrincipalStake;
    amountOfPrincipalStake += msg.value;
    IDepositContract(depositSystemContract).deposit{value: msg.value}(
      pubkey,
      //@audit Withdrawal credentials can point to a different address
      withdrawal_credentials,
      signature,
      deposit_data_root
    );

    emit NewAmountOfPrincipalStake(amountOfPrincipalStake, oldAmountOfPrincipalStake);
}
```

Recommendation(s): Revisit the protocol's design to avoid manipulations of internal accounting through arbitrary deposits.

Status: Fixed

Update from the client: Fixed in commit 5e2674955162c871e2b5a6e5d72a5c5ada8f3350

6.2 [Low] Possible reentrancy leads to a manipulated `fundsPendingWithdrawal` amount

File(s): `ObolValidatorManager.sol`

Description: The `_distributeFunds` function pays out principal before updating the `fundsPendingWithdrawal` state used by the PULL path. If the `principalRecipient` is a contract, it can re-enter during the principal payout and call `distributeFundsPull()`.

In this reentrant call, the contract observes the reduced balance but unchanged `fundsPendingWithdrawal`, causing it to allocate the remaining funds as pending withdrawals. Once control returns, the original PUSH call continues and transfers the reward payout as well.

As a result, `fundsPendingWithdrawal` is inflated without corresponding ETH in the contract. This breaks the invariant that `address(this).balance >= fundsPendingWithdrawal`, leading to incorrect accounting, potential withdrawal failures, and underflows in future distributions.

```
function _distributeFunds(uint256 pullOrPush) internal {
  /// ...

  //@audit external call before completing accounting update
  _payout(principalRecipient, _principalPayout, pullOrPush);

  // pay out reward
  _payout(rewardRecipient, _rewardPayout, pullOrPush);

  if (pullOrPush == PULL) {
    if (_principalPayout > 0 || _rewardPayout > 0) {
      // Write to storage
      fundsPendingWithdrawal = uint128(_memoryFundsPendingWithdrawal + _principalPayout + _rewardPayout);
    }
  }

  emit DistributeFunds(_principalPayout, _rewardPayout, pullOrPush);
}
```

Recommendation(s): Consider updating the state before the external payout calls.

Status: Fixed

Update from the client: Fixed in commit 5e2674955162c871e2b5a6e5d72a5c5ada8f3350

6.3 [Info] Arbitrary external calls in the `recoverFunds` function

File(s): `ObolValidatorManager.sol`

Description: The `recoverFunds` function is meant to allow users to recover non-OWR tokens from the contract. Although the `recipient` parameter of the function is checked, and it will be one of the addresses that the owner sets during deployment, having this permissionless function increases the attack surface.

A user can create a malicious `ERC20` token and send it to `ObolValidatorManager` contract of another user and then call the `recoverFunds` function.

This introduces two external calls on the malicious token contract where the attacker can execute arbitrary logic. Although in the current scope, there is no way to exploit this directly, adding a layer of access controls to the function is highly recommended to reduce the attack surface.

```
function recoverFunds(address nonOWRToken, address recipient) external payable {
//..
    //@audit first external call
uint256 amount = ERC20(nonOWRToken).balanceOf(address(this));
    //@audit second external call
nonOWRToken.safeTransfer(recipient, amount);
emit RecoverNonOWRecipientFunds(nonOWRToken, recipient, amount);
  }
```

Recommendation(s): Consider whitelisting tokens before they can be used in the function. Alternatively, restrict the function access to authorized actors only.

Status: Fixed

Update from the client: 0de5cb3ca566b189bccb057aee497e30c0def831

6.4 [Info] Call to arbitrary address in `withdraw`

File(s): `ObolValidatorManager.sol`

Description: The `withdraw` function retrieves the balance of a given `account` from the `pullBalances` mapping and transfers the funds using `account.safeTransferETH(amount)`. Since `account` is an arbitrary address supplied by the caller, the contract performs an external call to any address without restrictions.

```
function withdraw(address account) external {
    uint256 amount = pullBalances[account];
    unchecked {
      fundsPendingWithdrawal -= uint128(amount);
    }
    pullBalances[account] = 0;
    //@audit call to arbitrary address
    account.safeTransferETH(amount);

    emit Withdrawal(account, amount);
}
```

Recommendation(s): Implement a check to ensure that the transfer is only executed when `amount > 0`

Status: Fixed

Update from the client: Fixed in commit 5e2674955162c871e2b5a6e5d72a5c5ada8f3350

6.5 [Best Practices] Usage of hardcoded values

File(s): `ObolValidatorManager.sol`

Description: The `ObolValidatorManager` contract uses the hardcoded 48 value to check the public key length in multiple functions. It is best practice to avoid hardcoding values directly in the code. Instead, define a constant with a meaningful name that can be easily updated when necessary and improves the code's readability.

Recommendation(s): Consider defining a constant `PUBLIC_KEY_LENGTH` instead of using the hardcoded value in the code.

Status: Fixed

Update from the client: 0de5cb3ca566b189bccb057aee497e30c0def831

6.6 [Best practice] Functions are marked payable, but don't use `msg.value`

File(s): `ObolValidatorManager.sol`

Description: The `distributeFunds`, `distributeFundsPull`, and `recoverFunds` functions are marked payable, but throughout the body of the function, `msg.value` is never used. If users call these functions with `msg.value > 0`, those funds will either be immediately sent back to the user if the `PUSH` version is used, or stored in the contract as rewards that can be withdrawn immediately after if the `PULL` variant is used.

Similarly the `recoverFunds` is marked `payable` but doesn't use `msg.value` in any way. Users calling this function with `msg.value > 0` will end up donating their Ether to the contract.

Recommendation(s): Reconsider the need to have these functions `payable`.

Status: Fixed

Update from the client: 0de5cb3ca566b189bccb057aee497e30c0def831

6.7 [Best practice] Lack of input sanitization for the contract `owner`

File(s): `ObolValidatorManagerFactory.sol`

Description: The `createObolValidatorManager` function of the `ObolValidatorManagerFactory` contract takes an `address owner` as input parameter, but the function doesn't check if the `owner` is set to `address(0)`.

```
function createObolValidatorManager(
    address owner,
    address principalRecipient,
    address rewardRecipient,
    address recoveryAddress,
    uint64 principalThreshold
  ) external returns (ObolValidatorManager ovm) {
    // ...
    ovm = new ObolValidatorManager(
      consolidationSystemContract,
      withdrawalSystemContract,
      depositSystemContract,
        owner, //@audit can be address(0)
      principalRecipient,
      rewardRecipient,
      recoveryAddress,
      principalThreshold
    );
    //...
}
```

Recommendation(s): Consider adding a zero address check for the `owner` parameter.

Status: Fixed

Update from the client: 0de5cb3ca566b189bccb057aee497e30c0def831

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 Obol documentation

Obol's smart contract documentation is available in the README file within the project's GitHub repository, as well as in their online documentation.

The README file provides an overview of the system's general purpose and includes a section outlining the development dependencies. It also offers step-by-step instructions for building and testing the code.

The functions are thoroughly documented. Each function is accompanied by detailed comments that explain the logic behind its implementation, as well as the role of each parameter. Additional comments are provided for code sections containing more complex logic.

The Obol team also provided internal documentation that covers the design choices and key functions. All questions and concerns raised by the Nethermind Security team were addressed thoroughly.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge build
[] Compiling...
[] Compiling 74 files with Solc 0.8.19
[] Solc 0.8.19 finished in 21.45s
Compiler run successful with warnings:
Warning (3628): This contract has a payable fallback function, but no receive ether function. Consider adding a receive
↪  ether function.
  --> src/test/ovm/mocks/SystemContractMock.sol:13:1:
   |
13 | contract SystemContractMock {
   | ^ (Relevant source part starts here and spans across multiple lines).
Note: The payable fallback function is defined here.
  --> src/test/ovm/mocks/SystemContractMock.sol:53:3:
   |
53 |   fallback(bytes calldata) external payable returns (bytes memory) {
   |   ^ (Relevant source part starts here and spans across multiple lines).
```

8.2 Tests Output

```
Ran 7 tests for src/test/ovm/ObolValidatorManagerFactory.t.sol:ObolValidatorManagerFactoryTest
[PASS] testCan_createOWRecipient() (gas: 2785275)
[PASS] testCan_emitOnCreate() (gas: 2787216)
[PASS] testCannot_createWithInvalidRecipients() (gas: 22051)
[PASS] testCannot_createWithInvalidThreshold() (gas: 21895)
[PASS] testFuzzCan_createOWRecipient(uint64) (runs: 100, : 1401686, ~: 1401686)
[PASS] testFuzzCannot_CreateWithLargeThreshold(address,uint64) (runs: 100, : 14555, ~: 14555)
[PASS] testFuzzCannot_CreateWithZeroThreshold(address) (runs: 100, : 19183, ~: 19183)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 35.25ms (62.89ms CPU time)

Ran 30 tests for src/test/ovm/ObolValidatorManager.t.sol:ObolValidatorManagerTest
[PASS] testCan_OWRIsPayable() (gas: 60240)
[PASS] testCan_distributeDirectDepositsAsReward() (gas: 98126)
[PASS] testCan_distributeMultipleDepositsToPrincipalRecipient() (gas: 78753)
[PASS] testCan_distributeMultipleDepositsTorewardsRecipient() (gas: 73441)
[PASS] testCan_distributePushAndPull() (gas: 148949)
[PASS] testCan_distributeToBothRecipients() (gas: 121760)
[PASS] testCan_distributeToNoRecipients() (gas: 16575)
[PASS] testCan_distributeToPullFlow() (gas: 140505)
[PASS] testCan_distributeToSecondRecipient() (gas: 70873)
[PASS] testCan_emitOnDistributeToNoRecipients() (gas: 16385)
[PASS] testCan_recoverNonOWRFundsToRecipient() (gas: 171921)
[PASS] testCannot_distributeTooMuch() (gas: 86394)
[PASS] testCannot_recoverFundsToNonRecipient() (gas: 19162)
[PASS] testCannot_reenterOWR() (gas: 2092253)
[PASS] testCannot_requestConsolidation() (gas: 1040468476)
[PASS] testCannot_requestWithdrawal() (gas: 1040438232)
[PASS] testCannot_setPrincipalRecipient() (gas: 48916)
[PASS] testDefaultParameters() (gas: 17283)
[PASS] testDeposit() (gas: 53580)
[PASS] testFuzzCan_distributeDepositsToRecipients(uint64,uint8,uint256,uint256) (runs: 100, : 2174230, ~: 1876856)
[PASS] testFuzzCan_distributePullDepositsToRecipients(uint64,uint8,uint256,uint256) (runs: 100, : 2592076, ~: 2389951)
[PASS] testOwnerInitialization() (gas: 7781)
[PASS] testReceiveERC20() (gas: 37998)
[PASS] testReceiveETH() (gas: 12054)
[PASS] testReceiveTransfer() (gas: 12049)
[PASS] testRequestBatchConsolidation() (gas: 1434720)
[PASS] testRequestBatchWithdrawal() (gas: 1259962)
[PASS] testRequestSingleConsolidation() (gas: 233958)
[PASS] testRequestSingleWithdrawal() (gas: 197798)
[PASS] testSetPrincipalRecipient() (gas: 25054)
Suite result: ok. 30 passed; 0 failed; 0 skipped; finished in 254.22ms (316.97ms CPU time)
```

8.3 Automated Tools

8.3.1 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.