Security Review Report NM-0326 Canary

NETHERMIND SECURITY

(Oct 31, 2024)

Contents

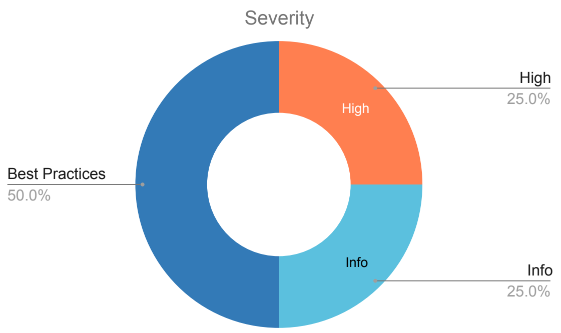
1 Executive Summary

This document presents the security review performed by Nethermind Security for the Canary staking vault contracts, which are designed to function as a restaking aggregator. The Canary team has developed two distinct vault contracts. The first vault manages ERC20 tokens, allowing administrators to deposit these assets into the Symbiotic protocol. The second vault focuses on managing ERC721 tokens.

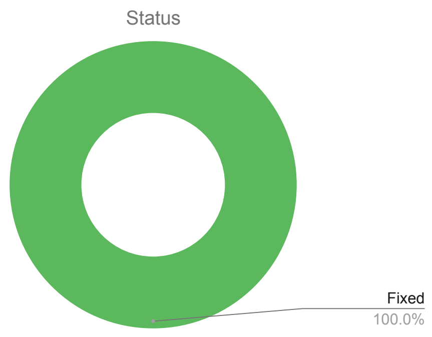
This security review focused on the current version of the protocol, which is still under development. Notably, some features, such as the integration with Symbiotic, remain incomplete. Therefore, a follow-up audit is recommended once the protocol is fully developed and additional functionalities are introduced.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** fifteen (4) points of attention, where one is classified as `High` and three 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 compilation, tests, and automated tests. Section 9 concludes the document.



(a)



(b)

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (1), **Best Practices** (2).
Distribution of status: Fixed (0), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (4)

Summary of the Audit

Audit Type	Security Review
Initial Report	Oct 17, 2024
Response from Client	Regular responses during audit engagement
Final Report	Oct 31, 2024
Repository	contracts
Commit (Audit)	696df5b544cbf217fbb83b444605f2af8930db02
Commit (Final)	a49e98aae1bf1e0893da28fdc599e55c10e30963
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	CanaryNFT.sol	81	4	4.9%	15	100
2	clToken.sol	18	3	16.7%	6	27
3	Vault.sol	210	13	6.2%	36	259
4	VaultManager.sol	100	6	6.0%	18	124
5	TokenFactory.sol	44	14	31.8%	16	74
6	NFTVault.sol	60	9	15.0%	19	88
7	NFTVaultManager.sol	53	5	9.4%	18	76
8	NFTFactory.sol	35	7	20.0%	9	51
	Total	**601**	**61**	**10.1%**	**137**	**799**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect assignment of access control roles in `Vault` and `NFTVault` contracts	High	Fixed
2	Unnecessary `safeApprove(...)` call in `withdrawFromSymbiotic(...)` function	Info	Fixed
3	Ensure the correct number of `clTokens` in the `Vault` contract	Best Practices	Fixed
4	Integrate `setWithdrawPeriod(...)` functionality into `initialize(...)` function	Best Practices	Fixed

4 System Overview

Canary has implemented two types of vault contracts responsible for managing ERC20 and ERC721 assets. The primary function of the ERC20 vault is to act as a restaking aggregator, connecting with various restaking options, such as Symbiotic.

The reviewed contracts have been categorized into three main components: factories and management contracts, receipt tokens, and vaults, which will be detailed in the following sections.

4.1 Factories and Management Contracts

The `NFTFactory` and `TokenFactory` contracts serve as factory contracts responsible for creating receipt tokens. Each receipt token is deployed via these factories as a proxy. The factory handles the proper initialization of these tokens and transfers the ownership of the proxy admin to the designated external owner.

The `VaultManager` and `NFTVaultManager` contracts operate similarly to the factories but are responsible for the complete initialization of vaults. Initially, they deploy all receipt tokens through the respective factories and subsequently deploy the vault itself, which is initialized by the manager. The vault is created as a proxy to enable upgradability. Furthermore, the `MINTER` role is assigned to the vault for all receipt tokens, ensuring that only the vault can mint these tokens.

4.2 Receipt Tokens

Receipt tokens come in two variants based on the ERC standard: `CanaryNFT`, which is minted as a receipt for deposited NFTs in the `NFTVault`, and `clToken`, which is minted upon depositing ERC20 assets into the `Vault` contract.

Each receipt token has a designated withdrawal period, functioning as a notice period before withdrawing the underlying asset from the vault. Each `Vault` manages three distinct `clToken` types, each associated with different withdrawal periods: 7 days, 30 days, and 90 days. The `NFTVault`, however, issues only a single type of CanaryNFT with a 7-day withdrawal period.

All receipt tokens are also pausable, allowing the protocol to halt the transfer of these tokens if needed.

4.3 Vaults

The vaults are categorized into two types: the `Vault` contract, responsible for managing ERC20 assets, and the `NFTVault` for handling ERC721 assets. Each vault is designed to manage only one underlying token.

The vault architecture is consistent across both types. Users can deposit the underlying assets via the `stake(...)` function. For the `NFTVault`, users specify the `tokenId`, and in return, they receive a receipt token corresponding to the same `tokenId`. In the case of the `Vault`, users specify the amount of ERC20 tokens they wish to deposit along with the `index` of the `clToken` they want to receive as receive.

```
// NFTVault.sol function:
function stake(uint256 tokenId) external;

// Vault.sol function:
function stake(uint256 underlyingAmount, uint256 index) external;
```

When a user decides to withdraw their underlying asset, they must call the `requestWithdraw(...)` function, which initiates a withdrawal request and burns the user's receipt token. Once the withdrawal period associated with the receipt token has elapsed, the user can finalize the process by calling the `claimWithdraw(...)` function to retrieve their asset.

```
// NFTVault.sol functions:
function requestWithdraw(uint256 tokenId) external;

function claimWithdraw(uint256 tokenId) external;

// Vault.sol functions:
function requestWithdraw(uint256 amount, uint256 index) external;

function claimWithdraw(uint256 index) external;
```

The `Vault` contract is designed to function as a restaking aggregator, meaning it has the capability to interact with restaking protocols. To facilitate this, the vault introduces two functions that allow administrators with the `SYMBIOTIC_ROLE` to deposit and withdraw assets from the Symbiotic vault.

```
function depositIntoSymbiotic(uint256 amount) public onlyRole(SYMBIOTIC_ROLE)

function withdrawFromSymbiotic(uint256 amount) public onlyRole(SYMBIOTIC_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 [High] Incorrect assignment of access control roles in `Vault` and `NFTVault` contracts

File(s): `Vault.sol`, `NFTVault.sol`

Description: The `Vault` and `NFTVault` contracts inherit from `AccessControlEnumerableUpgradeable`, which implements a role-based access control model. Critical roles, including `DEFAULT_ADMIN_ROLE` for both contracts and `SYMBIOTIC_ROLE` in the `Vault` contract, are assigned during the `initialize(...)` function. These roles are automatically granted to the `msg.sender`.

However, because the vaults are instantiated and initialized by a manager contract, the `msg.sender` at the time of the `initialize(...)` call is the manager contract itself. As a result, the privileged roles are inadvertently assigned to the manager contract, rather than an external account. This misconfiguration prevents privileged accounts from accessing or using certain privileged functions (e.g., `depositIntoSymbiotic(...)`).

Recommendation(s): Ensure that privileged roles are correctly assigned to external accounts.

Status: Unresolved

6.2 [Info] Unnecessary `safeApprove(...)` call in `withdrawFromSymbiotic(...)` function

File(s): `Vault.sol`

Description: In the `withdrawFromSymbiotic(...)` function, the `Vault` withdraws its underlying assets from the Symbiotic vault. The current implementation includes `safeApprove(...)` call before invoking the `withdraw(...)` function:

```
function withdrawFromSymbiotic(uint256 amount) public onlyRole(SYMBIOTIC_ROLE) {
   // ...
   // Approve symbioticVault to spend the vault's shares
   symbioticVault_.safeApprove(address(symbioticVault_), amount); // @audit can be deleted

   symbioticVault_.withdraw(address(this), amount);
   // ...
}
```

The `withdraw(...)` function in the Symbiotic vault contract (Default Collateral Contract) does not require approval. It burns the user's shares directly via its internal `_burn(...)` function, making the `safeApprove(...)` call redundant.

Recommendation(s): Remove the unnecessary `safeApprove(...)` call to avoid redundant operations.

Status: Unresolved

6.3 [Best Practices] Ensure the correct number of `clTokens` in the `Vault` contract

File(s): `Vault.sol`

Description: The `Vault` contracts are designed to maintain exactly three `clTokens` with different withdrawal periods: 7 days, 30 days, and 90 days. However, the current check in the `initialize(...)` function only verifies that at least one token is provided:

```
if (_clTokens.length == 0) {
    revert AtLeastOneCLTokenAddressMustBeProvided();
}
```

This check does not enforce the intended requirement of having exactly three tokens.

Recommendation(s): Update the check in the `initialize(...)` function to ensure that exactly three `clTokens` are provided during initialization.

Status: Unresolved

6.4 [Best Practices] Integrate `setWithdrawPeriod(...)` functionality into `initialize(...)` function

File(s): `CanaryNFT.sol`, `clToken.sol`

Description: The `setWithdrawPeriod(...)` function is designed to set the `withdrawPeriod` during the deployment process, and it is called only once in factory contracts. Since this value is only set once, it would be more efficient to handle the assignment directly within the `initialize(...)` function.

In its current design, the `setWithdrawPeriod(...)` function can still be called on the implementation contract. While this doesn't introduce a security risk, it creates conflicts with the intent of the `_disableInitializers()` function, which is meant to prevent initialization.

Recommendation(s): Consider removing the `setWithdrawPeriod(...)` function and incorporating the assignment of the `withdrawPeriod` within the `initialize(...)` function.

Status: Unresolved

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 Canary documentation

The **Canary** contracts lack comprehensive documentation and NatSpec comments in some parts of the code, which made certain sections challenging to understand. However, the **Canary** team effectively addressed the concerns and questions raised by the **Nethermind Security** team during regular calls.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 102 files with Solc 0.8.25
[] Solc 0.8.25 finished in 5.23s
Compiler run successful!
```

8.2 Tests Output

```
> forge test --fork-url https://mainnet.infura.io/v3/<api-key>
Ran 12 tests for test/NFTVault.t.sol:NFTVaultTest
[PASS] testCannotClaimBeforeWithdrawalPeriod() (gas: 230990)
[PASS] testClaimWithdraw() (gas: 254608)
[PASS] testClaimWithdrawBeforeRequest() (gas: 73297)
[PASS] testClaimWithdrawSameTokenIdTwice() (gas: 254118)
[PASS] testDeployNFTVaultWithDifferentUnderlying() (gas: 7833906)
[PASS] testDeployNFTVaultWithSameUnderlying() (gas: 16627)
[PASS] testRequestWithdraw() (gas: 228354)
[PASS] testRequestWithdrawByInvalidOwner() (gas: 187288)
[PASS] testRequestWithdrawMultipleTokens() (gas: 372736)
[PASS] testRequestWithdrawSameTokenIdTwice() (gas: 228843)
[PASS] testStake() (gas: 200421)
[PASS] testStakeSameTokenIdTwice() (gas: 185923)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 2.81s (1.89s CPU time)

Ran 17 tests for test/Vault.t.sol:VaultTest
[PASS] testCannotUpdateWithdrawPeriod() (gas: 17626)
[PASS] testCannotWithdrawOrClaimOthersTokens() (gas: 568156)
[PASS] testClaimWithdraw() (gas: 231308)
[PASS] testClaimWithdrawBeforePeriod() (gas: 237916)
[PASS] testClaimWithdrawWithoutRequest() (gas: 177224)
[PASS] testCreateNewVaultWithDifferentDefaultOperator() (gas: 15357785)
[PASS] testRequestWithdraw() (gas: 240245)
[PASS] testRequestWithdrawInsufficientBalance() (gas: 166255)
[PASS] testRequestWithdrawInsufficientDelegation() (gas: 169090)
[PASS] testSetDefaultOperator() (gas: 27957)
[PASS] testSetDefaultOperatorByAnyAddress() (gas: 44947)
[PASS] testSetInvalidStakeLimit() (gas: 180285)
[PASS] testSetStakeLimit() (gas: 31811)
[PASS] testStake() (gas: 176804)
[PASS] testStakeInsufficientBalance() (gas: 65919)
[PASS] testStakeLimitReached() (gas: 51807)
[PASS] testStakeZeroAmount() (gas: 26310)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 5.03s (1.19s CPU time)
```

```
Ran 19 tests for test/Vault.fork.t.sol:VaultTest
[PASS] testCannotUpdateWithdrawPeriod() (gas: 17627)
[PASS] testCannotWithdrawOrClaimOthersTokens() (gas: 664165)
[PASS] testClaimWithdraw() (gas: 471824)
[PASS] testClaimWithdrawBeforePeriod() (gas: 439641)
[PASS] testClaimWithdrawWithoutRequest() (gas: 376187)
[PASS] testCreateNewVaultWithDifferentDefaultOperator() (gas: 15357785)
[PASS] testDepositIntoSymbiotic() (gas: 323459)
[PASS] testRequestWithdraw() (gas: 441499)
[PASS] testRequestWithdrawInsufficientBalance() (gas: 366373)
[PASS] testRequestWithdrawInsufficientDelegation() (gas: 368643)
[PASS] testSetDefaultOperator() (gas: 27958)
[PASS] testSetDefaultOperatorByAnyAddress() (gas: 44991)
[PASS] testSetInvalidStakeLimit() (gas: 377599)
[PASS] testSetStakeLimit() (gas: 31812)
[PASS] testStake() (gas: 375868)
[PASS] testStakeInsufficientBalance() (gas: 79319)
[PASS] testStakeLimitReached() (gas: 65433)
[PASS] testStakeZeroAmount() (gas: 26288)
[PASS] testWithdrawFromSymbiotic() (gas: 353666)
Suite result: ok. 19 passed; 0 failed; 0 skipped; finished in 5.03s (18.52s CPU time)

Ran 3 test suites in 5.95s (12.87s CPU time): 48 tests passed, 0 failed, 0 skipped (48 total tests)
```

Remarks about Canary test suite

The **Canary** team has developed a testing suite that covers the basic flows of the protocol, including deposits, withdrawals, and various privileged actions. However, the protocol should always ensure that user impersonation is properly handled in the tests, as incorrect implementation of these tests led to a severe issue within the protocol.

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