Security Review Report
NM-0579 Celo Contracts 13

NETHERMIND SECURITY

(July 25, 2025)

Contents

1 Executive Summary

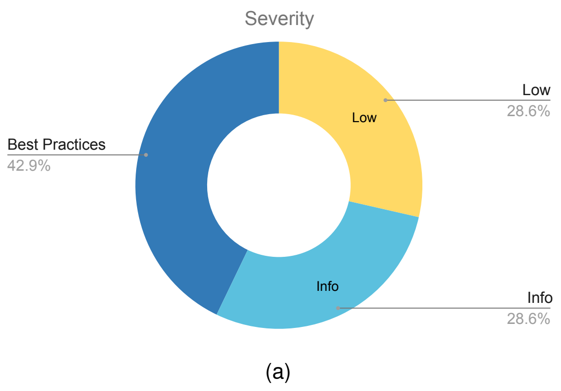
This document presents the results of a security review conducted by Nethermind Security for the Celo core-contracts V13 release. The core-contracts facilitate Celo features such as validator and group management, epoch rewards and governance. The changes in the V13 release will be applied to the currently deployed V12 contracts.

The changes include replacement of L1/L2 specific features with general functions to facilitate the transition to an Optimism based network, removal of BLS based validator signer signatures, and deprecation of some unused features.This review focuses on the changes applied to the Celo `core-contracts/13` compared to the `core-contracts/12` release.

The audit comprises 21 Solidity files. **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 seven points of attention, with two being classified as `Low`, and four being classified as `Info` 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 test output. Section 9 concludes the document.



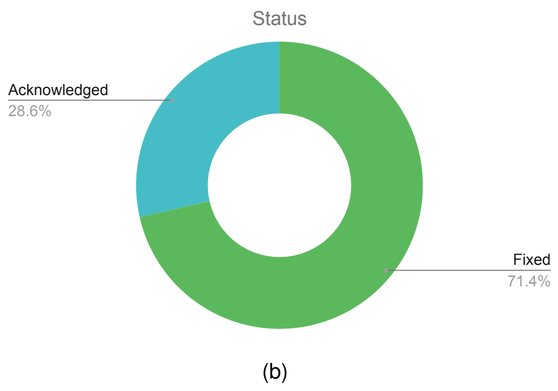


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (2), **Best Practices** (3).
Distribution of status: Fixed (5), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	July 16, 2025
Final Report	July 25, 2025
Repository	`celo-org/celo-monorepo`
Initial Commit	`807c5cc77aa380ce120ba3e4761f82de08f6a635`
Final Commit	`89fa8f2660c0dde368cbbceddb0f25b77ffd7e1c`
Documentation	`Cel2 Specification`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/common/GoldToken.sol	128	120	93.8%	33	281
2	contracts/common/Accounts.sol	589	418	71.0%	107	1114
3	contracts/common/Permissioned.sol	7	4	57.1%	1	12
4	contracts/common/interfaces/IAccounts.sol	41	1	2.4%	8	50
5	contracts/common/interfaces/ICeloToken.sol	10	5	50.0%	1	16
6	contracts/governance/Governance.sol	1041	488	46.9%	167	1696
7	contracts/governance/EpochRewards.sol	345	159	46.1%	35	539
8	contracts/governance/ReleaseGold.sol	429	264	61.5%	73	766
9	contracts/governance/Election.sol	715	422	59.0%	103	1240
10	contracts/governance/GovernanceSlasher.sol	90	44	48.9%	18	152
11	contracts/governance/interfaces/IReleaseGold.sol	40	5	12.5%	5	50
12	contracts/governance/interfaces/IElection.sol	103	4	3.9%	4	111
13	contracts/governance/interfaces/IEpochRewards.sol	12	1	8.3%	1	14
14	contracts/governance/interfaces/IValidators.sol	67	5	7.5%	6	78
15	contracts/governance/interfaces/ILockedCelo.sol	36	1	2.8%	6	43
16	contracts-0.8/common/CeloUnreleasedTreasury.sol	48	36	75.0%	14	98
17	contracts-0.8/common/EpochManager.sol	500	221	44.2%	102	823
18	contracts-0.8/common/interfaces/IWETH.sol	10	1	10.0%	6	17
19	contracts-0.8/common/interfaces/IStandardBridge.sol	11	16	145.5%	1	28
20	contracts-0.8/governance/Validators.sol	718	411	57.2%	94	1223
21	contracts-0.8/governance/interfaces/IValidatorsInitializer.sol	20	1	5.0%	2	23
	Total	**4960**	**2627**	**53.0%**	**787**	**8374**

3 Summary of Issues

	Finding	Severity	Update
1	An elected validator that deregisters can permanently DOS the `EpochManager`	Low	Fixed
2	`EpochManager` oracle can diverge from address in registry	Low	Acknowledged
3	A Validator that deaffiliates can permanently DOS the EpochManager	Info	Fixed
4	Burned `GoldToken` amount is measured inconsistently	Info	Fixed
5	Function visibility for `totalSupply` can be updated	Best Practices	Fixed
6	Solidity style guides and naming patterns	Best Practices	Fixed
7	`GoldToken` storage variable `totalSupply_` can be marked deprecated	Best Practices	Acknowledged

4 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.

5 Issues

5.1 [Low] An elected validator that deregisters can permanently DOS the `EpochManager`

File(s): `/packages/protocol/contracts-0.8/governance/Validators.sol`

Description: Validators have a permissionless path to deregister by calling the `deregisterValidator` function of the `Validators` contract. If one of the elected validators calls `deregisterValidator()` after the system has been initialized, but before new epoch processing started or if an elected validator deregisters in between two epochs, it will lead to a permanent DOS of the `EpochManager` contract.

Consider the following scenario: Epoch 1 is finalized and the system picks 5 new Validators as `electedAccounts`. One validator out of these 5 deregisters by calling the `deregisterValidator()` function on the `Validators` contract.

When the `startNextEpochProcess()` function will be called, inside the `allocateValidatorsRewards()` function, the `EpochManager` calls the `computeEpochReward()` function which will revert because of the `require(isValidator(account), “Not a validator”);` check. This will cause any call to `startNextEpochProcess` to fail, halting the Epoch processing flow.

Recommendation(s): Consider adding extra checks such that elected validators would not be allowed to deregister.

Status: Fixed

Update from the client: Fixed in PR#11443.

5.2 [Low] `EpochManager` oracle can diverge from address in registry

File(s): `packages/protocol/contracts-0.8/common/EpochManager.sol`

Description: The `EpochManager` contract tracks the oracle using a public storage variable `oracleAddress`. The oracle is also tracked using the registry contract. Upon initialization the storage variable will be set based on the registry oracle address.

```
function initialize(...) external initializer {
  // ...
  setOracleAddress(registry.getAddressForOrDie(SORTED_ORACLES_REGISTRY_ID));
}
```

However, the `EpochManager` contract also exposes a function `setOracleAddress` which accepts an arbitrary oracle address. This allows the oracle used by the `EpochManager` to be different from the oracle specified by the registry:

```
function setOracleAddress(address newOracleAddress) public onlyOwner {
  require(newOracleAddress != address(0), "...");
  require(newOracleAddress != oracleAddress, "...");
  require(!isOnEpochProcess(), "...");
  oracleAddress = newOracleAddress;
  emit OracleAddressSet(newOracleAddress);
}
```

The design purpose and expectations of a registry contract is that addresses only need to be changed at the registry, for all other dependent smart contracts to also reflect this change. By storing the oracle address in the `EpochManager` contract directly and exposing a setter, this convention is broken and allows for the `EpochManager` address to deviate from what is tracked on the registry. The oracle addresses can deviate in two scenarios:

- The oracle address is changed at the registry, this doesn't change the oracle used by the `EpochManager` causing them to be different;
- The oracle address is changed at the `EpochManager`, while the rest of the protocol still uses the oracle address tracked by the registry;

Recommendation(s): Consider changing `EpochManager` to use the registry for getting the oracle address instead of using a storage variable and setter.

Status: Acknowledged

Update from the client: Will not be fixed, but in next releases the `Initializer` will be updated, tracked in this issue: Issue#11452.

5.3 [Info] A Validator that deaffiliates can permanently DOS the EpochManager

File(s): /packages/protocol/contracts-0.8/governance/Validators.sol

Description: Upon initializing the system via initializeSystem the admin provides a list of firstElected addresses that will be stored in the electedAccounts array and based on the length of this array the electedSigners will also be set.

If one of these Validators calls the deaffiliate() function before startNextEpochProcess() is being called, the EpochManager contract will be permanently DOS.

Picture the following scenario. The system is initialized with electedAccounts having a length of 5. One Validator out of these 5 deaffiliates by calling the deaffiliate() function on the Validators contract.

When the startNextEpochProcess() function will be called, inside the allocateValidatorsRewards() function, the EpochManager calls the computeEpochReward() function with the following input params:

```
uint256 validatorReward = validators.computeEpochReward(
electedAccounts[i],
validatorScore,
_epochProcessing.perValidatorReward
);
```

This call is part of a for loop that will run until we reach electedAccounts.length which is still 5. When the loop will reach the unaffiliated Validator address, the call will revert on this line of code:

```

function computeEpochReward(...) external view virtual returns (uint256) {
    //..
    //@audit this function returns 0x0 and the check fails
    address group = getMembershipInLastEpoch(account);
    require(group != address(0), "Validator not registered with a group");
    //..
    //..
  }
```

This will prevent anyone from ever starting the Epoch processing flow.

Recommendation(s): Consider starting the Epoch processing right after the system is initialized in order to minimize the chances of this scenario from happening.

Status: Fixed

Update from the client: Fixed in PR#11443.

5.4 [Info] Burned `GoldToken` amount is measured inconsistently

File(s): packages/protocol/contracts/common/GoldToken.sol

Description: The function getBurnedAmount is used to determine the amount of assets that have been burned, where assets are considered burned if they have been sent to the 0xdead address. There is another function circulatingSupply which has a comment stating it calculates the total supply of tokens excluding those held by the burn address.

However, these functions determine the burned amount of tokens differently. getBurnedAmount only considers assets sent to the 0xdead address as burned, while circulatingSupply considers assets both from 0xdead and 0x0:

```
// Considers `0xdead`
function getBurnedAmount() public view returns (uint256) {
  return balanceOf(BURN_ADDRESS);
}

// Considers `0xdead` and `0x0`
function circulatingSupply() external view returns (uint256) {
  return allocatedSupply().sub(getBurnedAmount()).sub(balanceOf(address(0)));
}
```

Recommendation(s): Consider changing getBurnedAmount to account for both the 0xdead and 0x0 address, and removing the now-unnecessary 0x0 balance check in circulatingSupply.

Status: Fixed

Update from the client: Fixed in PR#11454.

5.5 [Best Practices] Function visibility for `totalSupply` can be updated

File(s): `packages/protocol/contracts/common/GoldToken.sol`

Description: In the previous V12 contracts, the function `totalSupply` was used by `allocatedSupply` in the case of an L1 call. With the V13 contract changes, this logic branch has been removed and `totalSupply` is no longer called internally by the contract:

```
function allocatedSupply() public view returns (uint256) {
  if (isL2()) {
    return CELO_SUPPLY_CAP - getCeloUnreleasedTreasury()
      .getRemainingBalanceToRelease();
  } else {
    return totalSupply(); // This branch is now removed
                          // `totalSupply` no longer called internally
  }
}
```

Recommendation(s): Since `totalSupply` is no longer being called internally, the visibility of this function can be changed from `public` to `external`.

Status: Fixed

Update from the client: Fixed in PR#11451.

5.6 [Best Practices] `GoldToken` storage variable `totalSupply_` can be marked deprecated

File(s): `packages/protocol/contracts/common/GoldToken.sol`

Description: As part of transitioning Celo to an Optimism L2, the Celo token conditional L1/L2 logic has been removed in the `GoldToken` contract. Particularly, the function `totalSupply` now directly returns the total supply based on the fixed supply cap, rather than using the internal storage variable `totalSupply_`:

```
// Code before V13 change
function totalSupply() public view returns (uint256) {
  if (isL2()) {
    return CELO_SUPPLY_CAP;
  } else {
    return totalSupply_; // This branch is now removed
  }
}
```

As a result, the `totalSupply_` storage variable is no longer used. However, unlike other storage variables that have been deprecated with the V13 changes, its name has not been changed to include a “deprecated” suffix.

Recommendation(s): For consistency with other deprecated storage variables, consider marking `totalSupply_` as deprecated.

Status: Fixed

Update from the client: Fixed in PR#11451.

5.7 [Best practices] Solidity style guides and naming patterns

File(s): /packages/protocol/contracts-0.8/common/EpochManager.sol

Description: The allocateValidatorsRewards function is marked internal but is not prefixed with an underscore. The name should be _allocateValidatorsRewards in order to respect Solidity's style guides.

There is also a slight deviation in the naming pattern for some getter and helper functions. All getter function names in EpochManager feature the prefix "get" except for numberOfElectedInCurrentSet which breaks this convention. Another group of functions use the prefix "is", and

We also noticed a slight deviation in the naming pattern when it comes to the getters and other helper functions. There are groups of "getter" functions or "is" functions that retrieve storage fields or query the status of the protocol, respectively. There are two functions that break this naming pattern:

```
function getCurrentEpoch()
function getEpochByNumber(...)
function getEpochByBlockNumber(...)
function getEpochNumberOfBlock(...)

function numberOfElectedInCurrentSet(...) // Breaks naming convention
                                          // `getNumberOfElectedInCurrentSet`
```

```
function isBlocked(...)
function isTimeForNextEpoch(...)
function isOnEpochProcess(...)

function systemAlreadyInitialized(...) // Breaks naming convention
                                       // `isSystemAlreadyInitialized`
```

Recommendation(s): Consider renaming the functions:

- numberOfElectedINCurrentset to getNumberOfElectedINCurrentset;
- systemAlreadyInitialized to isSystemAlreadyInitialized().

If this change is implemented, these functions must be renamed also in the IEpochManager interface.

Status: Acknowledged

Update from the client: As the contracts are deployed and people may be using the already available names, we note this issue but will not react at this point.

6 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 Celo documentation

The `Celo` team has provided a comprehensive walkthrough of the contracts in scope. A specification for the expected behavior of the contracts was provided. Moreover, the team addressed the questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

Documentation describing the overall purpose of the smart contracts has also been provided in the `Cel2 Specification`.

7 Test Suite Evaluation

7.1 Compilation Output

```
dev@machine protocol % forge build
[] Compiling...
[] Compiling 158 files with Solc 0.8.19
[] Compiling 229 files with Solc 0.5.17
[] Solc 0.8.19 finished in 30.36s
[] Solc 0.5.17 finished in 122.23s
Compiler run successful
```

7.2 Tests Output

```
user@machine protocol % git log -1
commit 0b169743a2a0b4c02e778bbd2ca026ee63742f48
(HEAD, tag: core-contracts.v13.post-audit, origin/release/core-contracts/13, release/core-contracts/13)

user@machine protocol % cd packages/protocol; forge test
Ran 402 test suites in 2.11s (8.05s CPU time): 2092 tests passed, 0 failed, 0 skipped (2092 total tests)
```

8 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