Security Review Report
NM-0422 Settle

NETHERMIND SECURITY

(Feb 27, 2025)

Contents

1 Executive Summary

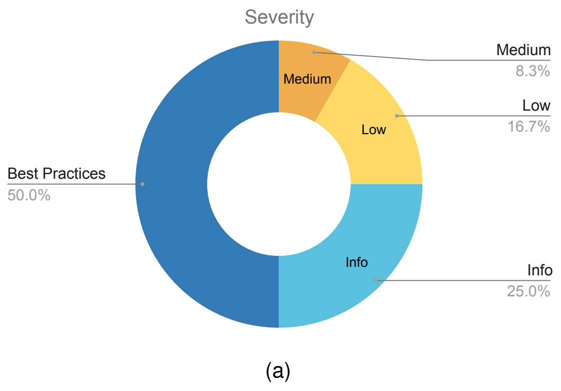
This document presents the security review performed by Nethermind Security for Settle smart contracts. Settle is a stable coin managed by manual mint and burn mechanisms.

Management operations are protected behind a multisig wallet, which requires approval by a minimum number of signers. Minted tokens are sent to a vault, where other roles can distribute them as appropriate.

The audit comprises 249 lines of solidity code, including three main contracts: **MultiSigWallet**, **Vault**, and **StableToken**.

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** twelve points of attention, where one is classified as `Medium`, two are classified as `Low`, and nine 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.



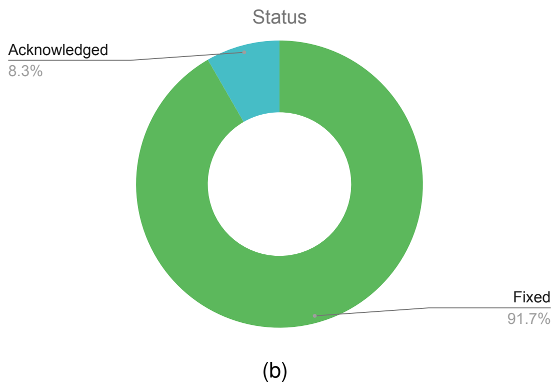


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (2), **Undetermined** (0), **Informational** (3), **Best Practices** (6).
Distribution of status: Fixed (11), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	January 30, 2025
Final Report	February 27. 2025
Repository	settle-token-solidity
Commit	9e0ea0643a38afd1d2b2034a2cd48a7a9fc634bf
Final Commit	312f5a421034b4aca87becbb72d32b5d72280832
Documentation	Private
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/MultiSigWallet.sol	167	3	1.8%	39	209
2	contracts/Vault.sol	31	1	3.2%	8	40
3	contracts/StableToken.sol	51	2	3.9%	12	65
	Total	**249**	**6**	**2.4%**	**59**	**314**

3 Summary of Issues

	Finding	Severity	Update
1	`MultiSigWallet::removeOwner()` does not remove addresses from the `owners` array	Medium	Fixed
2	Submited transactions do not have a deadline	Low	Fixed
3	The `getOwners(...)` function may return invalid results	Low	Fixed
4	Removing owners does not remove their confirmations	Info	Fixed
5	The `burn` function is `internal` and it seems that the code can't be reached	Info	Fixed
6	`isConfirmed` mapping is not update when removing an owner	Info	Acknowledged
7	Inheriting `Ownable` and `AccessControl` may be redundant	Best Practices	Fixed
8	The `getTransaction(...)` function returns values for non-existent transactions	Best Practices	Fixed
9	`ERC20Pausable` extension is being disabled	Best Practices	Fixed
10	`onlyContract(...)` modifier is not used	Best Practices	Fixed
11	Lack of enforcement for `msg.value` inside the `receive()` function	Best Practices	Fixed
12	Minting and burning operations are not affected by pause state	Best Practices	Fixed

4 System Overview

The Settle protocol implements a stable token using a multisig for management operations. The system consists of three main components working together:

- Multi-Signature Wallet (MultiSigWallet.sol)
- Stable Token (StableToken.sol)
- Token Vault (Vault.sol)

StableToken are minted through the multisig wallet. Usually, the minted tokens will go to the **Vault** for later distribution. Access control for the **Vault** differs from multisig to make the distribution process easier.

4.1 MultiSigWallet.sol

The contract is a secure multi-signature wallet that requires multiple confirmations from authorized owners to execute transactions. It implements the following features:

- Manages a list of owners and their confirmation requirements for transaction execution.
- Supports transaction submission, confirmation, revocation, and execution workflows.
- Allows for dynamic owner management and confirmation requirement adjustments through contract-level functions.

4.2 StableToken.sol

The contract is an ERC20-compliant token with additional features from OpenZeppelin. It includes burn, pause, and permit functionality for enhanced token management. The contract also supports standard token operations like `transfer(...)` and `transferFrom(...)` with pause protection.

4.3 Vault.sol

The contract provides secure storage and management of the stable token. Tokens are expected to be sent here after minting operations for distribution. It implements role-based access control for the different vault operations.

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] `MultiSigWallet::removeOwner()` does not remove addresses from the `owners` array

File(s): `contracts/MultiSigWallet.sol`

Description: The `removeOwner()` function inside `MultiSigWallet` doesn't remove addresses from the `owners` array. This can lead to a situation where there will be more owners in the `owners` array than the owner addresses tracked by the `isOwner` mapping. Let's call the later group "active owners".

Picture the following scenario.

– We initialize the `MultiSigWallet` with 5 owners and `numConfirmationsRequired = 3`;

The contract's state will look like this

```
index  [  0,     1,     2,     3,     4   ]
owners [addr1, addr2, addr3, addr4, addr5 ]
// this is set to true for all the addresses above
mapping(address => bool) public isOwner;
```

– We call `addOwner` 5 times to add 5 new addresses (i.e `addr6 -> addr10`);
– We call `removeOwner` 5 times to remove the initial 5 addresses;

The contract's state will look like this

```
index  [  0,     1,     2,     3,     4,     5,     6,     7,     8,     9    ]
owners [addr1, addr2, addr3, addr4, addr5, addr6, addr7, addr8, addr9, addr10 ]
// this is set to true only for addresses from 6 -> 10
mapping(address => bool) public isOwner;
```

– We call `changeRequirement` and set `numConfirmationsRequired = 7`;

Since this functions checks that `_numConfirmationsRequired <= owners.length`, the check will pass, even though we only have 5 active owners that can confirm a transaction which are tracked by the `isOwner` mapping.

This will lead to a permanent DoS of the `MultiSigWallet` because the threshold for executing transactions will never be met again.

Even if the current owners attempt to add a new owner, this will not work because in order to execute this transaction, they will need 7 confirmations now, but they can get 5 at most.

Recommendation(s): When an owner is removed from the multisig, their address should be removed from the `owners` array as well.

Status: Fixed.

Update from the client: We added the functionality to remove from the owners array the address has been revoked as a multisig owner. Fixed in ea65f9e8.

6.2 [Low] Submited transactions do not have a deadline

File(s): `contracts/MultiSigWallet.sol`

Description: The `MultiSigWallet` contract allows owners to submit actions in the way of `Transactions`.

```
struct Transaction {
    address to;
    uint256 value;
    bytes data;
    bool executed;
    uint256 numConfirmations;
}
```

These actions can be executed after the threshold of confirmations is reached. The `Transaction` concept lacks of a mechanism to make a transaction stale. A `Transaction` always wait to be confirmed and cannot be never invalidated. `Transactions` could be executed long time ago after they were submited causing their results to be very different to what was expected the time of the submission.

Recommendation(s): Add a deadline field to `Transactions` so they are not always valid waiting for confirmations.

Status: Fixed.

Update from the client: The property `deadline` was added to the transaction structure. Fixed in commit ea65f9e8.

6.3 [Low] The `getOwners(...)` function may return invalid results

File(s): `contracts/MultiSigWallet.sol`

Description: The `getOwners(...)` function returns the `owners` arrays. The action of removing an owner does not remove the owner from the `owners` array. Because of this, after owners are removed, the `getOwners(...)` function will return accounts that may not be owners anymore. The function can also return duplicated value if an owner was removed and added again.

Recommendation(s): Reconsider the need of having the `owners` array and enumerating the `owners`.

Status: Fixed.

Update from the client: We changed the name of the function to `getActiveOwners()`. After adding the functionality to remove inactive addresses from the `owners` array, this function returns only active owners. Fixed in commit 41bbde8f.

6.4 [Info] Removing owners does not remove their confirmations

File(s): `contracts/MultiSigWallet.sol`

Description: When owners are removed they past confirmations are still valid. It is possible to remove an owner that confirmed a transaction that has not been executed yet and the confirmation will remain valid.

Recommendation(s): Consider if this behavior is desired and document it.

Status: Fixed.

Update from the client: We included the functionality to remove the confirmations when removing a multisig owner.

Update from Nethermind: The current change only accounts for transactions submitted by the owner being removed. It does not check for transactions confirmed by the designated owner.

Update from the client: Fixed in commit e146dc.

6.5 [Info] The `burn` function is `internal` and it seems that the code can't be reached

File(s): `contracts/StableToken.sol`

Description: The `StableToken` contract implements a `burn` function that is currently marked with `internal` visibility. The function is not called by any other function, making it unreachable.

```
    function burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");
        super._burn(account, amount);
    }
```

Recommendation(s): Consider removing the function or making it accessible.

Status: Fixed.

Update from the client: The function `burn` was removed. Fixed in commit fb0414a8.

6.6 [Info] `isConfirmed` mapping is not update when removing an owner

File(s): `contracts/MultiSigWallet.sol`

Description: The `removeOwner(...)` removes an onwer from the multisig and removes all the current active confirmations of the owner. However, the function does not update the `isConfirmed` mapping while removing the confirmation. This result in an inconsistent state for the contract

Recommendation(s): Consider updating the `isConfirmed` mapping when removing owners.

Status: Acknowledged.

6.7 [Best Practice] Inheriting `Ownable` and `AccessControl` may be redundant

File(s): `contracts/Vault.sol`

Description: The `Vault` contract inherits both `Ownable` and `AccessControl`. Both inherited contracts has similar goal but with different implementations. While `Ownable` offers a more simple access control mechanism based on one account, `AccessControl` offers more granular and flexible mechanisms. The functionalities providen by `AccessControl` can be seen as a superset of the ones provided by `Ownable`. Inheriting the two contracts may be unnecessary. In the current state, all the roles are assigned to the same account.

Recommendation(s): Reconsider the desired access control design to simplify the contract.

Status: Fixed.

Update from the client: The `Ownable` library was removed. Fixed in commit f334f696.

6.8 [Best Practice] The `getTransaction(...)` function returns values for non-existent transactions

File(s): `contracts/MultiSigWallet.sol`

Description: The `getTransaction(...)` function returns the details of the `Transaction` identified by the provided transaction index. If the provided index is not used by any `Transaction` it returns `zero` for all the fields of a `Transaction`. The same index may return a different result later when it is used for a `Transaction`.

Recommendation(s): Consider making the `getTransaction(...)` function to revert if the provided index is not being used.

Status: Fixed.

Update from the client: A validation to the transaction index parameter has been included. Fixed in commit a8d283e8.

6.9 [Best Practice] `ERC20Pausable` extension is being disabled

File(s): `contracts/StableToken.sol`

Description: The `ERC20Pausable` extension allows to pause all the functionalities from a token by overriding the `_update(...)` function. The `StableToken` contracts inherits the `ERC20Pausable` extension, but overrides the `_update(...)` function too. This causes the effects from the `ERC20Pausable` extensions to be removed.

Recommendation(s): Consider usign the mechanism provided by the `ERC20Pausable` extension or remove it and use only `Pausable` if it is more convenient.

Status: Fixed.

Update from the client: The extension `ERC20Pausable` was removed. Fixed in commit a647b10a.

6.10 [Best Practice] `onlyContract(...)` modifier is not used

File(s): `contracts/Vault.sol`

Description: The `onlyContract(...)` modified declared in the `Vault` contract is not used through the contract.

Recommendation(s): Consider removing the code as it is not used.

Status: Fixed.

Update from the client: The modifier has been removed. Fixed in commit 97a4b6e3.

6.11 [Best practice] Lack of enforcement for `msg.value` inside the `receive()`function

File(s): `contracts/MultiSigWallet.sol`

Description: The `receive()` function has no checks for `msg.value`. This allows any user to spam this function with `msg.value = 0` in order to emit worthless events.

Recommendation(s): Consider emitting the event only if `msg.value > 0`.

Status: Fixed.

Update from the client: We added the code to prevent `msg.value` equal to zero. Fixed in commit 1f7eb283.

6.12 [Best practice] Minting and burning operations are not affected by pause state

File(s): `contracts/StableToken.sol`

Description: Because it inherits `ERC20Pausable`, the `StableToken` contract could be paused, but the `mint(...)` and `burn(...)` functions do not use the `whenNotPaused` modifier. The admin might want to pause all token operations in case of emergency, but in the current implementation only token transfers will be paused.

Recommendation(s): Consider adding the `whenNotPaused` modifier on the `mint` and `burn` functions as well.

Status: Fixed.

Update from the client: The modifier `whenNotPaused` was added only to the `mint` function. Fixed in commit 307c47e7.

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 the Settle Protocol documentation

The Settle Protocol provided a document containing the protocol's architecture and the expected main interactions. Alongside the document, the Settle team explained multiple details for the implementation to the Nethermind team.

8 Complementary Checks

8.1 Compilation Output

```
> npx hardhat compile
Generating typings for: 3 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 38 typings!
Compiled 3 Solidity files successfully (evm target: paris).
```

8.2 Tests Output

```
npx hardhat test
Compiled 30 Solidity files successfully (evm target: paris).


  MultiSigWallet
    should be deployed with correct owners and requirement
    should allow an owner to submit a tx
    should prevent a non-owner to submit a tx
    should allow an owner to confirm a tx
    should prevent a non-owner from confirming a tx
    should execute a tx when enough confirmations are gathered
    should execute a tx when more than the required confirmations
    should prevent execution if not enough confirmations
    should allow an owner to revoke a confirmation
    should prevent a non-owner from revoking a confirmation
    should only allow multisig to add additional owners
    should verify transaction properties

  STT Token Minting
    should verify new token owner
    should execute goverment tx
    should allow multisig to add a new owner
    should allow removing an owner
    should not allow removing an owner if number becomes less than required
    should mint tokens

  Vaul Testing
    should verify multisig as vault owner
    should validate the vault owner role
    should validate the default admin role
    should validate the token address admin role
    should validate vault balance
    should validate only vault owner can transfer
    should validate transfer receipient
    should validate that vault owner can transfer
    should validate the access control roles
    should validate that a not-owner account cannot create a new role
    should validate the granting of a new role
    should be possible to revoke role
    should validate that the new owner can transfer funds

  StableToken
    Should have the correct name and symbol
    Should have an initial supply of 0
    Should revert if trying to transfer to the zero address
    Should revert if trying to transfer from the zero address
    Should revert if trying to transfer more tokens than the balance
    Should revert if trying to burn more tokens than the balance
    Should revert if trying to transfer ownership to the zero address
    Should allow transfers
    Should allow burning
    Should allow pausing and unpausing
    Should revert if a non-owner tries to pause or unpause
    Should allow the owner to transfer ownership


  43 passing (3s)
```

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