Security Review Report
NM-0447 Token Table

NETHERMIND
SECURITY

(Mar 21, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for Token Table's smart contracts written in Cairo programming languaged deployed on Starknet. The audit focused on pull request #3 which included bumping up Cairo version to 2.9.2, added Chainlink-related interfaces.

The audit comprises 1051 lines of code written in Cairo language. The audit focused on changes to the `unlocker`, `chainlink`, and `futureToken` 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** 6 points of attention, where two are classified as `Medium`, and four 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.

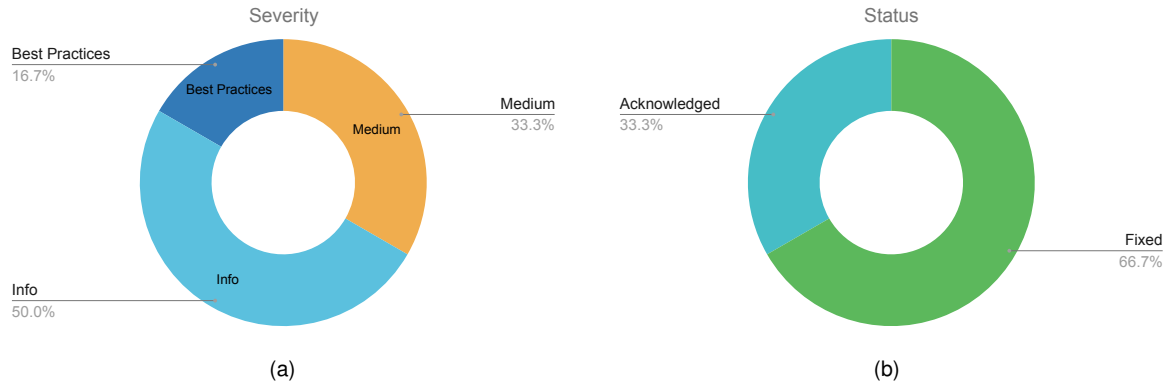

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (0), **Undetermined** (0), **Informational** (3), **Best Practices** (1). **Distribution of status: Fixed** (4), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 17, 2025
Response from Client	Regular responses during audit engagement
Final Report	March 21, 2025
Repository	tokentable-v2-starknet
Commit	30808fd
Final Commit	e3ec394
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/unlocker.cairo	894	35	3%	47	976
2	src/futuretoken.cairo	97	17	17%	12	126
3	components/structs/chainlink.cairo	60	7	11%	10	77
	Total	**1051**	**59**	**5.6%**	**69**	**1179**

3 Summary of Issues

	Finding	Severity	Update
1	Unbounded loop leading to index out of bound error prevents claiming of tokens	Medium	Fixed
2	The `amount_skipped` is not accounted for when chainlink price feed is set	Medium	Fixed
3	Chainlink round data manipulation in `claim` allows user to potentially claim more tokens than intended	Info	Acknowledged
4	One tolerance may not fit for all the unlock timestamps	Info	Acknowledged
5	Unreached contract state in `get_fee(...)` function	Info	Fixed
6	Inconsistency between `claim` and `delegate_claim`	Best Practices	Fixed

4 System Overview

`TokenTable` protocol is an on-chain token distribution product designed to streamline token ownership registration and distribution through smart contracts. The significant component in the TokenTable protocol is the `Unlocker` contract, which is designed for fine-token unlocking for smaller groups of recipients, such as treasury management and token unlocks. The changes in the TokenTable protocol include the following components:

- **chainlink.cairo** - Addition of `chainlink` smart contract interfaces and functions
- **unlocker.cairo** - Introduced integration mechanism with `Chainlink` oracle and major changes to `calculate_amount_claimable(...)` `simulate_amount_claimable(...)` together with `_update_actual_and_send(...)` functions.

The audited smart contract suite implements a modular and upgradable framework for managing token unlocking schedules and future token issuance on StarkNet. The design splits responsibilities across several specialized components, allowing each module to be upgraded or replaced independently. This architecture enhances maintainability and minimizes risk by isolating critical functions.

4.1 TTUnlocker.cairo

The `TTUnlocker` contract is the centerpiece of the system's vesting and unlock mechanism. It orchestrates token unlocks through a two-layered approach:

- **Presets and Actuals**: Administrators configure unlock schedules (presets) that specify time-based unlock parameters—such as linear start and end timestamps, unlock proportions (bips), and the number of unlock steps. Actual unlock instances (actuals) are then created based on these presets, determining when and how much of the token balance becomes claimable. The team introduced changes to `calculate_amount_claimable(...)` `simulate_amount_claimable(...)` and `_update_actual_and_send(...)` functions which the contract uses to calculate, send and update claimable amounts.
- **Integrated Controls and Hooks**: `TTUnlocker` implements features like cancellation, delegated claiming, and hook callbacks that allow external modules to react to events (e.g., preset creation or token claims). In addition, it incorporates Chainlink price feeds for scenarios where USD-based calculations are required. Chainlink integration was an additional feature introduced during the code update.

4.2 TTFutureToken.cairo

Acting as the representation of future token rights, the `TTFutureToken` contract is implemented as an `ERC721` token with additional functionality:

- **Minting control**: Only an authorized minter can invoke the mint function, thereby coupling token issuance to the underlying unlock mechanism.
- **Metadata and transfer management**: The contract extends standard `ERC721` functionality by integrating token metadata management and configurable transfer settings, ensuring that future tokens accurately reflect the evolving state of token unlocks.

The project removed the `get_claim_info(...)` function in the new version.

4.3 Chainlink.cairo

The `chainlink.cairo` contract is the Chainlink price feed integration to the TokenTable Protocol. The `chainlink.cairo` contract allowed the protocol to query price, get feed address from serialized data, and serialize chainlink feed address.

- **price query** - The contract introduced `get_price_at(...)` function to get token price at a given chainlink round id.

```
fn get_price_at(feed: ContractAddress, round_id: felt252) -> (u8, u128, u64);
```

- **chainlink feed address serialization** - The `serialize_chainlink_feed_address(...)` function allowed the contract to serialize contract addresses.

```
fn serialize_chainlink_feed_address(feed_address: ContractAddress) -> Span<felt252>;
```

- **get feed address from serialized data** - The `get_feed_address_from_serialized_data (...)` function allowed the Chainlink contract to get feed address from serialized data.

```
fn serialize_chainlink_feed_address(feed_address: ContractAddress) -> Span<felt252>;
```

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] Unbounded loop leading to index out of bound error prevents claiming of tokens

File(s): src/unlocker.cairo

Description: The `simulate_amount_claimable(...)` calls `simulate_get_expanded_actual_claim_timestamps_and_bips(...)` which returns `expanded_claim_timestamps_array` that contains all the unlock timestamps. It then loops through `expanded_claim_timestamps_array` to set the start and end indexes. It breaks when the timestamp at the current index is greater than or equal to the current timestamp, and this index will be the end index. The issue is that if the current timestamp is greater than the last timestamp at `expanded_claim_timestamps_array`, then it will not stop at the last index but will access the element beyond that which would lead to an index out-of-bound error, preventing users from claiming their tokens.

```
fn simulate_amount_claimable(
 self: @ContractState, actual_start_timestamp_absolute: u64,
 preset_linear_end_timestamp_relative: u64,
 ///.............................
 /// ........ Existing code ......
 ///.............................


 /// Expand linear timestamps so each unlock timestamp is explicitly represented
 let (expanded_claim_timestamps_array, expanded_claim_bips_array) = self
 .simulate_get_expanded_actual_claim_timestamps_and_bips(
 preset_linear_start_timestamps_relative,
 preset_linear_end_timestamp_relative,
 actual_start_timestamp_absolute,
 preset_num_of_unlocks_for_each_linear,
 preset_linear_bips,
 );
 /// Validate if round ID timestamps are within an acceptable time range
 let mut expanded_claim_timestamp_start_index = Bounded::MAX;
 let mut expanded_claim_timestamp_end_index =
 0; /// The most recent index that's not in the future, used as an endpoint for claim calculation
 /// We can perform calculation in this loop first, then validate round
 /// timestamps after. We can save one loop.
 loop {
 // @audit The expanded_claim_timestamp_end_index can be greater than the last index of the array, leading to index
  ↪  out-of-bound error
 /// `expanded_claim_timestamp_end_index` is used as `i` here
 let current_iteration_timestamp = *expanded_claim_timestamps_array
 .at(expanded_claim_timestamp_end_index);
 if expanded_claim_timestamp_start_index == Bounded::MAX
 && current_iteration_timestamp > actuals_chainlink_last_claimed_timestamp {
 expanded_claim_timestamp_start_index = expanded_claim_timestamp_end_index;
 }
 if current_iteration_timestamp >= claim_timestamp_absolute {
 break;
 }
 expanded_claim_timestamp_end_index += 1;
 };
 } else {
 }
}
```

Recommendation(s): Consider breaking if all the elements are iterated.

Status: Fixed

Update from the client: Fixed in commit 2e1f7f4.

Update from Nethermind Security: There's an issue with the fix that allows users to claim an extra unlock timestamp. For example, if `expanded_claim_timestamps_array = [100, 200, 300]` and `claim_timestamp_absolute = 201`, and since there's a check to break if `current_iteration_timestamp >= claim_timestamp_absolute`, the loop will break when `current_iteration_timestamp` is `300`. As a result, `expanded_claim_timestamp_end_index` will be `2` (the last index), allowing the user to claim an extra unlock timestamp.

Update from the client: Fixed in commit e3ec394.

6.2 [Medium] The `amount_skipped` is not accounted for when chainlink price feed is set

File(s): src/unlocker.cairo

Description: The `amount_skipped` is used while creating an `Actual` to allow the owner to set the amount claimed by the recipient. This will be accounted for when recipients claim the tokens by ensuring they will not get any tokens until `amount_skipped` is covered and only get the amount beyond it. This might be useful when the recipient has already received this amount from external sources, and the owner wants to account for this amount.

Thus, to account for this, when creating an `Actual`, the `amount_claimed` is set to `amount_skipped`. In the case when Chainlink price feed is disabled while claiming tokens, recipients get the token amount equal to the difference between `token_amount_claimable_from_start` and `amount_claimed`. However, when Chainlink price feed is enabled, the calculation for the amount of tokens the recipient would receive does not account for `amount_skipped`.

Recommendation(s): Consider modifying the calculation for the amount of tokens the recipient would receive in the case when Chainlink price feed is enabled to also account for `amount_skipped`.

Status: Fixed

Update from the client:The client disabled `amount_skipped` when using Chainlink in commit 035c704

6.3 [Info] Chainlink round data manipulation in `claim` allows user to potentially claim more tokens than intended

File(s): src/unlocker.cairo

Description: In the `TTUnlocker` contract's branch for schedules using Chainlink (in `simulate_amount_claimable`), the calculation of the claimable token amount depends on an array of round IDs supplied by the caller (via the parameter `extra_chainlink_round_ids`). Because these round IDs are used directly without independent validation, a user can try to supply "favorable" round IDs.

Check this code:

```
let round_id = *extra_chainlink_round_ids.at(i);
let round_data = chainlink_instance.round_data(round_id);
...
token_amount_claimable_from_last_claim += (*expanded_claim_bips_array.at(i))
 .into() * actual_total_amount
 * base.pow(chainlink_instance.decimals().into())
//@audit This is a user-provided input param, and the only check performed is that it is within the time tolerance limit
 / round_data.answer.into()
 / BIPS_PRECISION.into();
```

Imagine that a user who is entitled to claim supplies an `extra_chainlink_round_ids` array with round IDs pointing to rounds where the reported price (`round_data.answer`) is lower. This would increase the computed token amount claimable because the division by a lower number will inflate the result.

Imagine that the tolerance limit is set to 5 minutes, and the token's price jumps 50% from 10$ -> 15$ within 5 minutes. The user can call the `claim(...)` function and provide the `round_ids` from when the token's price was still 10$, enabling them to claim more tokens than intended.

Recommendation(s): Consider using Chainlink's defined method of fetching historical data then validating the rounds with the fetched historical data.

Status: Acknowledged.

Update from the client: Acknowledged, but no fix.

6.4 [Info] One tolerance may not fit for all the unlock timestamps

File(s): src/unlocker.cairo

Description: While claiming tokens, in the case of Chainlink price feed is enabled, the users have to provide round IDs for each unlock timestamp. The timestamps corresponding to the round IDs should be under the tolerance limit w.r.t the corresponding unlock timestamps. The owner is supposed to set the tolerance limit to a lower value. Otherwise, users may take advantage of it by providing round IDs corresponding to the lowest price within the tolerance limit, thus claiming more tokens. However, setting it to a lower value might not fit all the unlock timestamps, as Chainlink price feed data might not be available within the tolerance limit for some timestamps. Plus, in a volatile market, the price may drop significantly for a shorter duration and again become stable. Still, users may use the round ID corresponding to that price drop event if that fits within the tolerance limit, getting a significant advantage.

Recommendation(s): Consider using Chainlink's defined method of fetching historical data then validating the rounds with the fetched historical data. Another way would be to consider the last few prices within the tolerance limit and average them so that one significant price drop for a smaller duration may not manipulate the token amount.

Status: Acknowledged.

Update from the client: Acknowledged, but no fix.

6.5 [Info] Unreached contract state in `get_fee(...)` function

File(s): src/feecollector.cairo

Description: The `get_fee(...)` function which is used in calculating the amount of fees on the amount of tokens to transfer to the fee collector checks if the `custom_fee_bips` is set. If the `custom_fee_bips` is zero, it uses the `default_fee_bips`. In the event the `fee_bips` is set equal to the `BIPS_PRECISION`, the `fee_bips` used is zero as seen below:

```
fn get_fee(
 self: @ContractState, unlocker_instance: ContractAddress, tokens_transferred: u256
 ) -> u256 {
 let mut fee_bips = self.custom_fee_bips.read(unlocker_instance);
 if fee_bips == 0 {
 // If the fee is unset, the default fee should be used.
 fee_bips = self.default_fee_bips.read();
 } else if fee_bips == BIPS_PRECISION { //@audit this state will not be reached given MAX_FEE check in setting custom
  ↪  fee and default
 // If the fee is set to 100%, it means the fee is zero.
 fee_bips = 0;
 }
 tokens_transferred * fee_bips / BIPS_PRECISION
 }
```

The issue is either `custom_fee_bips` or `default_fee_bips` can never be set to `BIPS_PRECISION` given the `MAX_FEE` check in both `set_default_fee(...)` and `set_custom_fee(...)`.

The constant `BIPS_PRECISION = 10000` and `MAX_FEE = 1000`.

Recommendation(s): Consider updating the fee configuration mechanism to allow setting the fee to `BIPS_PRECISION` for the contract to reach that state.

Status: Fixed.

Update from the client: Fixed in commit 2e1f7f4.

6.6 [Best practice] Inconsistency between `claim` and `delegate_claim`

File(s): src/unlocker.cairo

Description: The `claim(...)` function returns the `amount_claimed` as a `u256` parameter.

```
fn claim(
 ref self: TContractState,
 actual_id: u256,
 claim_to: ContractAddress,
 recipient_id: u64,
 extra_data: Span<felt252>,
 ) -> u256;
```

The `delegate_claim`, on the other hand, does not return the `amount_claimed`.

```
fn delegate_claim(
 ref self: TContractState, actual_id: u256, recipient_id: u64, extra_data: Span<felt252>,
 );
```

Recommendation(s): Consider returning the `amount_claimed` in the `delegate_claim(...` function to maintain consistency.

Status: Fixed

Update from the client: fixed in commit 2e1f7f4.

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 TokenTable Protocol documentation

The TokenTable Protocol team provided enough information about their TokenTable project. This included documentation about the audited pull request. Additionally, the team answered every question during meetings or through messages which ensured successful security review of their codebase.

8 Test Suite Evaluation

8.1 Compilation Output

```
> scarb build
Compiling snforge_scarb_plugin v0.36.0
↪  (git+https://github.com/foundry-rs/starknet-foundry?tag=v0.36.0#6af1722bad3dd98001285e76f43a83d8b38614de)
Finished `release` profile [optimized] target(s) in 0.10s
Compiling lib(tokentable_v2) tokentable_v2 v2.6.0(/Scarb.toml)
Finished `dev` profile target(s) in 43 seconds
```

8.2 Tests Output

```
snforge test

Collected 13 test(s) from tokentable_v2 package
Running 13 test(s) from tests/
[IGNORE] tokentable_v2_tests::integration_tests::expand_timestamps_test
[PASS] tokentable_v2_tests::integration_tests::unlocker_withdraw_test (gas: ~6145)
[PASS] tokentable_v2_tests::integration_tests::unlocker_claimable_calculation_unit_test (gas: ~3282)
[PASS] tokentable_v2_tests::integration_tests::unlocker_cancel_test_1 (gas: ~6087)
[PASS] tokentable_v2_tests::integration_tests::deployer_test (gas: ~3069)
[PASS] tokentable_v2_tests::integration_tests::unlocker_create_actual_test (gas: ~6763)
[PASS] tokentable_v2_tests::integration_tests::unlocker_create_preset_test (gas: ~5332)
[PASS] tokentable_v2_tests::integration_tests::unlocker_cancel_test_0 (gas: ~5941)
[PASS] tokentable_v2_tests::integration_tests::unlocker_cancelable_test (gas: ~6034)
[PASS] tokentable_v2_tests::integration_tests::unlocker_claim_test (gas: ~6516)
[PASS] tokentable_v2_tests::integration_tests::unlocker_delegate_claim_test (gas: ~6381)
Total claimed in USD: 1234074060; Original total amount: 1234567890; Delta ceiling: 1%
[PASS] tokentable_v2_tests::fork_tests::fork_test_strk_1 (gas: ~6088)
Total claimed in USD: 1234567881; Original total amount: 1234567890; Delta ceiling: 1%
[PASS] tokentable_v2_tests::fork_tests::fork_test_strk_0 (gas: ~8313)
Running 0 test(s) from src/
Tests: 12 passed, 0 failed, 0 skipped, 1 ignored, 0 filtered out
```

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.