Security Review Report
NM-0462 zkLend Recovery

NETHERMIND
SECURITY

(Mar 3, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security of the zkLend's recovery contracts. zkLend is a money-market protocol built on Starknet that enables permissionless and compliant lending and borrowing. The recovery contracts audited in this engagement are designed to facilitate the distribution of user assets and future recovery fund proceeds following the zkLend security incident.

The recovery mechanism utilizes a Merkle tree structure to track and validate claims. Each affected user is assigned either an immediately withdrawable portion of tokens and/or a number of contribution shares entitling them to future contributions. The contract enforces a two-step claim process where users initially register their claims with a Merkle proof via the `register_and_claim(...)` function. Subsequent claims are processed through the `claim(...)` function.

Additionally, the contract allows any party to contribute funds to the recovery pool via the `contribute(...)` function, ensuring that users with contribution shares can receive future disbursements.

The audit focused on evaluating the security and correctness of the recovery contract implementation, ensuring that the smart contracts uphold the intended asset distribution and access control mechanisms securely and effectively.

The audited code comprises 486 lines of code written in the Cairo language. The audit focused on the implementation of the recovery 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** five points of attention, one of which is classified as `Critical` severity, one is classified as `Low` severity and three are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

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

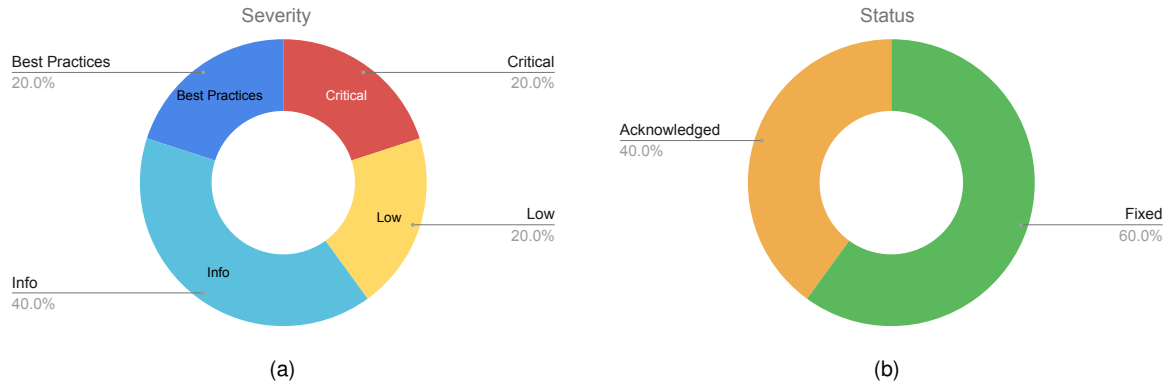

Fig. 1: Distribution of issues: Critical (1), **High** (0), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (2), **Best Practices** (1). **Distribution of status: Fixed** (3), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	February 28, 2025
Final Report	March 3, 2025
Repository	recovery
Audit Commit	4c58f99189dcbe300856652f6ef78ca737c3e7c4
Audit Class Hash	0x0146a0737a4197d8b4f753cec0f4ec5191242847f51f6f607b2a85470a04316a
Final Commit	abd4fb5b62028355ecbce1e6f99dd8f96dcd1281
Final Class Hash	0x04cc30bead28bc7c0c91589f00ab73ac06e9f547051a8a76a9c918b9eb6ab8e0
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/utils.cairo	34	5	14.7%	9	48
2	src/lib.cairo	452	96	21.2%	89	637
	Total	**486**	**101**	**20.8%**	**98**	**685**

3 Summary of Issues

	Finding	Severity	Update
1	Claim reusing allows for draining the funds	Critical	Fixed
2	Lack of checks for immediately withdrawable portion	Low	Acknowledged
3	Lack of owner ability to withdraw tokens	Info	Fixed
4	Some claims may be unreachable	Info	Acknowledged
5	Missing input validation	Best Practices	Fixed

4 System Overview

zkLend's recovery smart contract provides a structured method for distributing user assets and managing recovery fund proceeds following the zkLend security incident. The system ensures that each user receives an immediately withdrawable portion of tokens and/or contribution shares that entitle them to future token distributions. The complete allocation list is organized into a Merkle tree, enabling secure and efficient verification of claims.

4.1 Actors

- **Users**: Individuals affected by the incident who can claim their entitled tokens by interacting with the recovery contract.
- **Contributors**: Entities that add funds to the recovery pool, which are distributed proportionally among users with contribution shares.
- **Contract Owner**: The administrator who has the authority to pause/unpause the contract and add new contribution tokens.

4.2 Claiming Tokens

Users interact with the protocol through a "register-claim" process:

a. The first-time interaction requires calling `register_and_claim(...)` with an off-chain Merkle proof. This initiates withdrawal of the immediately available portion and registers the user's entitlement to future contributions. If available, users can also claim their share of any contribution token they specify.

b. Subsequent claims are made through the `claim(...)` function to receive additional distributed contribution tokens.

4.3 Contributing Funds

Anyone can contribute tokens to the recovery pool by calling the `contribute(...)` function. The only requirement is that the contribution amount must be greater than the minimum amount specified by the contract's owner.

4.4 Contract Functions

The recovery contract includes the following key functions:

- **register_and_claim(...)**: Registers the user with a Merkle proof and claims available tokens.
- **claim(...)**: Allows registered users to claim distributed contributions.
- **contribute(...)**: Enables contributors to add funds to the recovery pool.
- **pause(...)/unpause(...)**: Allows the contract owner to halt and resume operations.
- **add_contribution_token(...)**: Permits new tokens for contributions, managed by the contract owner.
- **get_all_contributions(...)**: Retrieves the total contribution amount for all contribution tokens.
- **get_claimable_contribution(...)**: Retrieves the claimable portion from contributions, provided the user has previously registered.

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 [Critical] Claim reusing allows for draining the funds

File(s): src/lib.cairo

Description: The `claim(...)` function may be abused to reuse the same claim multiple times, draining the contract from tokens. The `claim(...)` function takes `sources`, which may be a list of tokens chosen by the user. The function `get_claimable_contribution(...)` iterates over each token and calls `get_claimable_contribution_for_token(...)` to calculate the amount that should be sent to the user:

```
fn get_claimable_contribution_for_token(
    self: @ContractState,
    user: ContractAddress,
    user_share: u256,
    total_share: u256,
    token: ERC20ABIDispatcher,
) -> Option<TokenAmount> {
    let token_total_contribution: u256 = self
        .total_contributions
        .read(token.contract_address)
        .into();

    let user_amount: u128 = (token_total_contribution * user_share / total_share)
        .try_into()
        .unwrap();
    let claimed_amount = self.claimed_contributions.read((user, token.contract_address));

    if claimed_amount < user_amount {
        Option::Some(TokenAmount { token, amount: user_amount - claimed_amount })//@audit: this does not allow for small
         ↪  contributions payouts after some payout
    } else {
        Option::None
    }
}
```

Each call to `get_claimable_contribution_for_token(...)` would calculate the amount separately, and the `claimed_amount` is not updated in between. Next, the `get_claimable_contribution(...)` returns the `claimed` array, which updates the user's claimed amount and transfers the tokens. Malicious users can provide the same token multiple times, making the `get_claimable_contribution_for_token(...)` function return the same amount multiple times since the `claimed_amount` is not updated.

Test:

```
#[test]
fn test_claim_selected_duplicated_exploit() {
    let setup = setup_local();
    setup.accounts.owner.recovery.add_contribution_token(setup.contracts.token_a, 0);
    setup.accounts.owner.recovery.contribute(setup.contracts.token_a, 1_000);
    // Initial state:
    //      claimable tokens A:    1_000 (* decimals)
    //      withdrawable tokens A: 3_000 (* decimals)
    //      Charlie balance:       0 (* decimals)
    assert_eq!(
        setup.contracts.token_a.balance_of(setup.contracts.recovery.contract_address), 4_000,
    );
    // Charlie has claim of 1_000 shares of token A, where total shares is 2_000
    // and claimable token A balance is 1_000 (* decimals)
    // which should result 1_000 (* decimals) * 1_000 shares / 2_000 shares = 500 (* decimals)
    // But Charlie reuses the claim providing it eight times: 500 shares * 8
    // resulting in claiming 4_000 (* decimals) instead of 500 (* decimals) of tokens A,
    // stealing all claimable and withdrawable tokens
    // Note: Charlie has 0 withdrawable amount in the claim
    setup
        .accounts
        .charlie
        .recovery
        .register_and_claim(
            ContributionSource::Selected(array![
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
                setup.contracts.token_a, // 500 shares
            ].span()),
            setup.get_context(setup.accounts.charlie.address),
        );
    // After exploit state:
    //      claimable tokens A:    0 (* decimals)
    //      withdrawable tokens A: 0 (* decimals)
    //      Charlie balance:       4_000 (* decimals)
    assert_eq!(setup.contracts.token_a.balance_of(setup.accounts.charlie.address), 4_000);
    assert_eq!(
        setup.contracts.token_a.balance_of(setup.contracts.recovery.contract_address), 0,
    );
}
```

Recommendation(s): Consider updating `claimed_amount` after each amount calculation in `get_claimable_contribution_for_token(...)`. Alternatively, consider checking the `sources` array for duplicates.

Status: Fixed.

Update from the client: Fixed in commit `b8de2125a1522960f25d1e1e9831e6aed3e10666` by requiring selected tokens to be sorted and monotonically increasing.

6.2 [Low] Lack of checks for immediately withdrawable portion

File(s): `src/lib.cairo`

Description: The immediately withdrawable amounts defined in the users' claims are registered only by increasing `total_claimed_withdrawable`. However, there are no checks to ensure that the amount of directly transferred tokens equals all the withdrawable amounts in claims. If the sum of initially provided tokens by the team is not equal to the sum of each user's withdrawable amount, then it is possible that during the registration, the withdrawable amount would be taken from the contribution funds. That would make the contract's internal state incorrect since the `claimed_contributions` and `total_claimed_contribution` would not reflect the contributed amounts.

Recommendation(s): Consider ensuring that the immediately withdrawable funds equal the sum of all withdrawable amounts in claims. This can be achieved by storing transferred funds on chain, subtracting them during each registration, or ensuring this off-chain during contract deployment and funds provision.

Status: Acknowledged.

Update from the client: Will ensure the token amounts funded upon deployment match exactly with the Merkle tree. The Merkle tree is deterministically generated with a publicly accessible calculator, so the funding process is easily auditable.

6.3 [Info] Lack of owner ability to withdraw tokens

File(s): src/lib.cairo

Description: Some percentage of the tokens sent may never be claimed by the users. The current implementation does not allow the owner to withdraw the tokens. This may lead to the permanent freeze of tokens in the contract.

Recommendation(s): Consider implementing a functionality allowing the owner to withdraw the unclaimed tokens after the defined period to avoid freezing unclaimed funds.

Status: Fixed.

Update from the client: Added a time-locked owner withdrawal feature in commit `f04fe40796850c7f14d5ee61f5a388aa9dd86b21`.

6.4 [Info] Some claims may be unreachable

File(s): src/lib.cairo

Description: The list of addresses that are eligible for funds recovery is fetched from on-chain data and consists of addresses that held funds in zkLend at a certain period. However, some addresses may not be accounts but contracts used to manage funds and interact with zkLend functionalities. Those addresses may not have the possibility to interact with the recovery contract and may not be upgreadable. This would render some users unable to claim the funds.

Recommendation(s): Consider developing a strategy for the users who interacted with the zkLend through a contract. One possible solution is to fetch the deployer account address of a contract and allow this account to interact with the recovery contract instead. Note, however, that the deployer may not always be a user but may be the owner of a funds management protocol. Such cases should be considered, and cooperation may be needed between the protocol owners.

Status: Acknowledged.

Update from the client: After extensive profiling of recipient contract classes it's estimated that over 99.9% of the total value attribute to either account contracts or otherwise upgradeable contracts. The remaining are unknown classes but not necessarily non-upgradeable. It doesn't seem worth increasing attack surface for the extreme edge case.

6.5 [Best Practices] Missing input validation

File(s): src/lib.cairo

Description: Certain important state variables set by the contract owner lack input validation. Setting them to incorrect values will lead to unexpected contract behavior. Below is a list of places where additional checks could be implemented:

- `total_share` can be set to `0` in the constructor;
- `root` can be set to `0` in the constructor;
- `token` can be added with address `0` in `add_contribution_token(...)`;

Recommendation(s): Consider adding missing input validation in the abovementioned places.

Status: Fixed.

Update from the client: Added additional input validation in commit `abd4fb5b62028355ecbce1e6f99dd8f96dcd1281`.

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 zkLend's documentation

The documentation for the recovery contracts was provided through the README.md file as well as detailed code comments. This documentation provided a high-level overview of the protocol and details of its implementation. Moreover, the zkLend team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
Collected 31 test(s) from zklend_recovery package
Running 0 test(s) from src/
Running 31 test(s) from tests/
[PASS] zklend_recovery_tests::test_claim_withdrawable_can_only_be_claimed_once (gas: ~3183)
[PASS] zklend_recovery_tests::test_can_withdraw_after_unlock (gas: ~2885)
[PASS] zklend_recovery_tests::test_must_register_first (gas: ~2932)
[PASS] zklend_recovery_tests::test_anyone_can_contribute (gas: ~3419)
[PASS] zklend_recovery_tests::test_deployment (gas: ~2865)
[PASS] zklend_recovery_tests::test_cannot_add_duplicate_contribution_token (gas: ~3133)
[PASS] zklend_recovery_tests::test_claim_selected_contribution_tokens (gas: ~5054)
[PASS] zklend_recovery_tests::test_claim_withdrawable_only (gas: ~3626)
[PASS] zklend_recovery_tests::test_only_owner_can_pause (gas: ~2932)
[PASS] zklend_recovery_tests::test_only_owner_can_add_contribution_token (gas: ~2932)
[PASS] zklend_recovery_tests::test_only_owner_can_unpause (gas: ~3002)
[PASS] zklend_recovery_tests::test_token_transferred_on_contribution (gas: ~3834)
[PASS] zklend_recovery_tests::test_cannot_register_twice (gas: ~3209)
[PASS] zklend_recovery_tests::test_only_owner_can_withdraw (gas: ~2932)
[PASS] zklend_recovery_tests::test_claim_both_withdrawable_and_contribution (gas: ~6176)
[PASS] zklend_recovery_tests::test_cannot_register_with_empty_claim (gas: ~2933)
[PASS] zklend_recovery_tests::test_claim_contribution_can_only_be_claimed_once (gas: ~5283)
[PASS] zklend_recovery_tests::test_cannot_register_with_mismatched_recipient (gas: ~2934)
[PASS] zklend_recovery_tests::test_claim_multiple_tokens (gas: ~4800)
[PASS] zklend_recovery_tests::test_cannot_claim_duplicate_contribution_tokens_on_registration (gas: ~3606)
[PASS] zklend_recovery_tests::test_cannot_withdraw_before_unlock (gas: ~2934)
[PASS] zklend_recovery_tests::test_cannot_contribute_non_contribution_token (gas: ~2932)
[PASS] zklend_recovery_tests::test_cannot_contribute_when_paused (gas: ~2938)
[PASS] zklend_recovery_tests::test_cannot_claim_non_contribution_token (gas: ~3265)
[PASS] zklend_recovery_tests::test_get_all_contributions (gas: ~4156)
[PASS] zklend_recovery_tests::test_cannot_contribute_below_min_amount (gas: ~3197)
[PASS] zklend_recovery_tests::test_claim_empty_contribution (gas: ~3021)
[PASS] zklend_recovery_tests::test_cannot_register_when_paused (gas: ~2939)
[PASS] zklend_recovery_tests::test_cannot_claim_when_paused (gas: ~2938)
[PASS] zklend_recovery_tests::test_cannot_claim_duplicate_contribution_tokens (gas: ~3618)
[PASS] zklend_recovery_tests::test_cannot_register_with_invalid_proof (gas: ~2934)
Tests: 31 passed, 0 failed, 0 skipped, 0 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.