Security Review Report
NM-0369 Limit Orders
(Ekubo Extension)

NETHERMIND
SECURITY

(Jan 6, 2025)

Contents

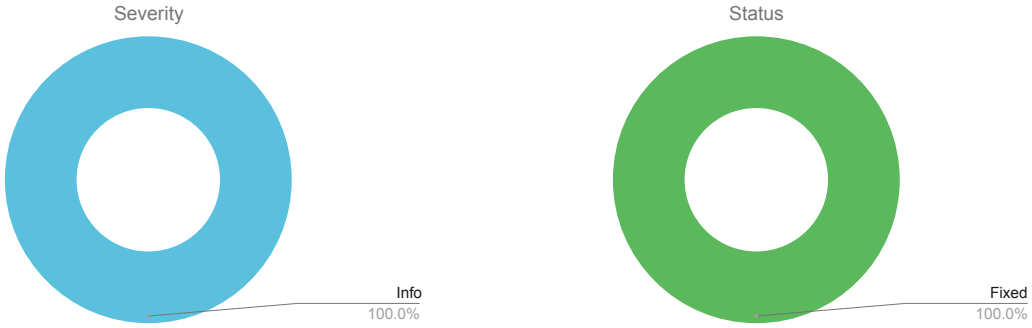
1 Executive Summary

This document outlines the security review conducted by Nethermind Security for LimitOrders extension in Ekubo. *Ekubo extensions* allow third-party developers to create new kinds of pools on *Ekubo* that integrate into the same ecosystem of aggregators and interfaces built on top of *Ekubo*.

Through the use of the *LimitOrder* extension, traders can create positions that are automatically withdrawn when executed.

The team conducted a six-engineer-week review of 568 lines of code. The `Ekubo` team has provided a thorough explanation of the code through multiple meetings. Aside from the provided explanation, the `Ekubo` and `Nethermind Security` engaged in multiple calls to clarify any remaining questions about the expected behavior of the protocol. **The audit was performed using**: (a) manual analysis of the codebase, and (b) writing test cases. **Along this document, we report** one point of attention classified as `Informational`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

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

Summary of the Audit

Audit Type	Security Review
Initial Report	Dec 9, 2024
Final Report	Jan 6, 2025
Methods	Manual Review, Tests
Repository	limit-orders-extension
Commit Hash	a1cbf7bdda33045819dbbe2f0feb8faa49795117
Final Commit Hash	a1cbf7bdda33045819dbbe2f0feb8faa49795117
Documentation	NatSpec and communication
Documentation Assessment	Medium
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/limit_orders.cairo	564	63	11.2%	82	709
2	src/lib.cairo	4	5	125.0%	3	12
	Total	**568**	**68**	**12.0%**	**85**	**721**

3 Summary of Issues

	Finding	Severity	Update
1	`_get_order_info(...)` function returns inconsistent results for non-existent orders	Info	Unresolved

4 System Overview

This section presents an overview of the *Limit Orders* extension.

4.1 Extensions

Ekubo implements the concept of Extensions, enabling users to add custom logic at certain pool lifecycle events. These customized pools can implement different features, such as limit orders. Ekubo allows Extensions to re-enter core contracts to set their own swaps, updates, etc. before/after specific actions are done.

As presented in the struct below, extensions are defined as part of the pool key in an immutable setting of the pool. This means that an extension is part of the identity of the pool, making pools with different extensions but the rest of settings equal to still be different pools

```
pub struct PoolKey {
    pub token0: ContractAddress,
    pub token1: ContractAddress,
    pub fee: u128,
    pub tick_spacing: u128,
    pub extension: ContractAddress,
}
```

As users choose pools to interact with, it is crucial to carefully evaluate the extensions involved, as they significantly impact the pool's lifecycle. Errors in extension implementation may lead to users losing funds.

An extension can implement eight different hooks to modify the pool's lifecycle:

- *before_initialize_pool(...)*
- *after_initialize_pool(...)*
- *before_swap(...)*
- *after_swap(...)*
- *before_update_position(...)*
- *after_update_position(...)*
- *before_collect_fees(...)*
- *after_collect_fees(...)*

4.2 LimitOrders

The **LimitOrders** extension allows users to sell one of the pool's tokens at a certain price. This is achieved by providing those tokens as liquidity at the specified price and automatically withdrawing the liquidity when the order is executed.

The extension implements three hooks:

- ***before_initialize_pool(...)*:** This hook is called every time a new pool is initialized using the extension. The implementation ensures that only the extension itself can initialize pools using the extension.
- ***after_swap(...)*:** The hook is called every time a swap is executed on a pool using the extension. The implementation of this hook must update any existing order. When an order is executed, the liquidity from it must be withdrawn, and the related tokens saved as balance in the Ekubo protocol. The hook updates the state of every order executed by the swap and the state of the extension for the pool triggering the hook.
- ***before_update_position(...)*:** This hook is triggered whenever a position in the pool is updated. The implementation ensures that only the extension can update positions from the pools using the extension. Users cannot manually manage positions in these pools.

4.3 Main actions

Three main flows can be triggered for the extension: **opening an order**, **closing an order**, and **executing orders**.

Open an order

To open an order, the user must **lock** the core contract from Ekubo and **forward** the lock to the extension using the **PlaceOrderForwardCallbackData** struct as data.

```
pub struct PlaceOrderForwardCallbackData {
    pub salt: felt252,
    pub order_key: OrderKey,
    pub liquidity: u128,
}
```

Through this data, the user specifies a **salt**, the **OrderKey**, and the **liquidity** to be provided (tokens to be sold). The **OrderKey** contains the information of which tokens are being sold, which tokens will be received in exchange and the price at which the tokens must be sold. Which of the two tokens from the pools will be sold is determined by the provided **tick**. **Ticks** divisible by **2 * tick_spacing** will be used to sell the **token0** from the pool, and the rest of the ticks will be used to sell the **token1** from the pool.

```
pub struct OrderKey {
    // The first token sorted by address
    pub token0: ContractAddress,
    // The second token sorted by address
    pub token1: ContractAddress,
    // The price at which the token should be bought/sold. Must be a multiple of tick spacing.
    // If the specified tick is evenly divisible by 2 * tick_spacing, it implies that the order is
    // selling token0. Otherwise, it is selling token1.
    pub tick: i129,
}
```

An order is identified by its **OrderKey**, its **salt**, and its **creator**. When an order is created, liquidity will be deposited to the pool, and a delta will be created that must be covered by the user before releasing the lock. Users cannot modify existing orders, create orders with zero liquidity, or create orders requiring a deposit of tokens different from the one the user wants to sell. The extension itself will manage all the liquidity deposited by creating orders. If multiple orders are created for the same range, their liquidity will be aggregated under the same position.

Users can use any pair of tokens they want. If no pool exists in Ekubo for the specified pair of tokens linked to the LimitOrders extension, it will be initialized, computing the starting price from the order.

Close an order

Users can close their orders at any point in time. Similar to the process for opening an order, the user must **lock** the core contract from Ekubo and **forward** the lock to the extension using the **CloseOrderForwardCallbackData** struct as data.

```
pub struct CloseOrderForwardCallbackData {
    pub salt: felt252,
    pub order_key: OrderKey,
}
```

The information from **CloseOrderForwardCallbackData**, together with the user executing the operation, contains all the details needed to identify the order.

An existing order may or may not have been executed. If the order was already executed, the liquidity related to it was already withdrawn from Ekubo, and the earned tokens were saved as balance belonging to the extension; in this case, the extension will load the number of tokens the user is entitled to from the liquidity provided earlier while creating the order. If the order has not been executed, the extension will update the position related to the order by removing the liquidity associated with the order. Multiple orders may be aggregated under the same position, so only the liquidity from the specific order must be withdrawn. Both of these actions, loading the tokens or removing liquidity, will create a delta of tokens that can be used/withdrawn by the user before removing the lock. During the process of closing an order, the state associated with it will also be cleared.

Executing orders

Orders are executed when the range where they are set is completely crossed by a swap. After every swap, the *after_swap(...)* hook from the extension is triggered. This hook will iterate over all the initialized ticks between the tick the pool was before the swap and the tick the pool was after the swap. Because liquidity can only be added to the pool by placing orders, every initialized tick corresponds to at least one active order. When an initialized tick is found, all the liquidity from the orders existing at that range is withdrawn because the orders were executed. The withdrawn tokens will be saved as a balance for the extension loaded after closing those orders. The iteration only executes orders for ticks that are not divisible by **2 * tick_spacing**, because every order is set in the range **(order_tick, order_tick + tick_spacing)**, it is known that every order has one tick that fits this property.

This hook also updates the state related to the pool where the swap was executed.

```
pub(crate) struct PoolState {
    // The number of times this pool has crossed an initialized tick plus one
    pub initialized_ticks_crossed: u64,
    // The last tick that was seen for the pool
    pub last_tick: i129,
}
```

The **initialized_ticks_crossed** value is used to mark the "time" when an order is created and check if it was executed later. The **last_tick** value stores the last tick processed by the extension; the value is updated after every swap executed on the pool.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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 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 formally pass to the client;

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 [Info] `_get_order_info(...)` function returns inconsistent results for non-existent orders

File(s): src/limit_orders.cairo

Description: The `_get_order_info(...)` function returns the information related to a specific order. The function returns a value of type `GetOrderInfoResult`.

```
pub struct GetOrderInfoResult {
    pub(crate) state: OrderState,
    pub executed: bool,
    pub amount0: u128,
    pub amount1: u128,
}
```

The `executed` value returned by this function is inconsistent for non-existing orders. The value of the `executed` field is defined by the result of the condition `initialized_ticks_crossed_at_order_tick > order.initialized_ticks_crossed_snapshot`. For non-existent orders, the `order.initialized_ticks_crosed_snapshot` is always zero, then the result of the expression `initialized_ticks_crossed_at_order_tick > order.initialized_ticks_crossed_snapshot` depends only on the `initialized_ticks_crossed_at_order_tick` value. The value `executed` for a non-existent order will be `true` if the tick was initialized and crossed at any point in time; the value of `executed` will be `false` if the tick was not initialized and crossed until now.

Recommendation(s): Consider keeping consistent the returned information for a non-existent order.

Status: Fixed.

Update from the client: The function behavior was changed to always return **executed: false** for orders with zero liquidity. The commit for this change was done in a repository different to the one audited.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. 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 LimitOrders extension documentation

The Ekubo team has provided documentation about how extensions integrate with the core protocol. Information about how the Limit Order extension should work was provided in the form of NatSpec and during meetings.

8 Test Suite Evaluation

8.1 Compilation Output

```
> scarb build
   Compiling snforge_scarb_plugin v0.1.0
    ↪  (git+https://github.com/foundry-rs/starknet-foundry.git?tag=v0.30.0#196f06b251926697c3d66800f2a93ae595e76496)
    Finished `release` profile [optimized] target(s) in 0.17s
   Compiling ekubo_limit_orders_extension v0.1.0 (.../limit-orders-extension/Scarb.toml)
    Finished `dev` profile target(s) in 7 seconds
```

8.2 Tests Output

```
> snforge test
   Compiling snforge_scarb_plugin v0.33.0
    ↪  (git+https://github.com/foundry-rs/starknet-foundry.git?tag=v0.33.0#221b1dbff42d650e9855afd4283508da8f8cacba)
    Finished `release` profile [optimized] target(s) in 0.40s
   Compiling test(ekubo_limit_orders_extension_unittest) ekubo_limit_orders_extension v0.1.0
    ↪  (.../EkuboProtocol/limit-orders-extension/Scarb.toml)
    Finished `dev` profile target(s) in 9 seconds


Collected 19 test(s) from ekubo_limit_orders_extension package
Running 19 test(s) from src/
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_constructor_sets_call_points (gas: ~1057)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_pool_is_not_initialized (gas: ~1056)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_pool_cannot_be_initialized_manually (gas: ~1056)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_cannot_place_order_for_zero_liquidity (gas: ~1186)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_cannot_close_order_for_non_existent_pool (gas: ~1190)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_max_liquidity_min_price_sell_token0 (gas:
↪  ~1415)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_cannot_place_two_orders_for_same_key (gas: ~2078)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_cannot_close_order_twice (gas: ~1768)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_max_liquidity_min_price_sell_token1 (gas:
↪  ~1878)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_and_partially_execute_sell_token1 (gas: ~2081)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_orders_only_one_executed_sell_token1 (gas: ~2933)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_max_liquidity_one_price_sell_token0 (gas:
↪  ~1944)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_max_liquidity_max_price_sell_token0 (gas:
↪  ~1947)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_orders_only_one_executed_sell_token0 (gas: ~2902)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_max_liquidity_max_price_sell_token1 (gas:
↪  ~1415)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_max_liquidity_one_price_sell_token1 (gas:
↪  ~1876)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_and_partially_execute_sell_token0 (gas: ~2046)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_and_fully_execute_sell_token0 (gas: ~2318)
[PASS] ekubo_limit_orders_extension::limit_orders_test::test_place_order_and_fully_execute_sell_token1 (gas: ~2390)
Tests: 19 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.