Security Review Report
NM-0418 CCDM

NETHERMIND
SECURITY

(Jan 20, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security of the Cross-chain Deposit Module (CCDM). The review focused on a new feature introduced to the CCDM protocol, enabling deposits on the source chain to be included in a Merkle tree. This enhancement optimizes gas costs by bridging only the Merkle root to the destination chain instead of explicitly sending individual depositor data, such as addresses and deposit amounts. Depositors can later prove their deposits on the destination chain using Merkle proofs to withdraw their funds. This feature aligns with CCDM's goal of facilitating gas-efficient cross-chain deposit campaigns and was assessed for its impact on the protocol's security and functionality.

The audited code comprises 1321 lines of code written in the Solidity language. The audit focused on the new Merkle deposit feature introduced to the CCDM system.

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** two points of attention, one of which is classified as `Critical` severity and one is classified as `Medium` severity. 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.

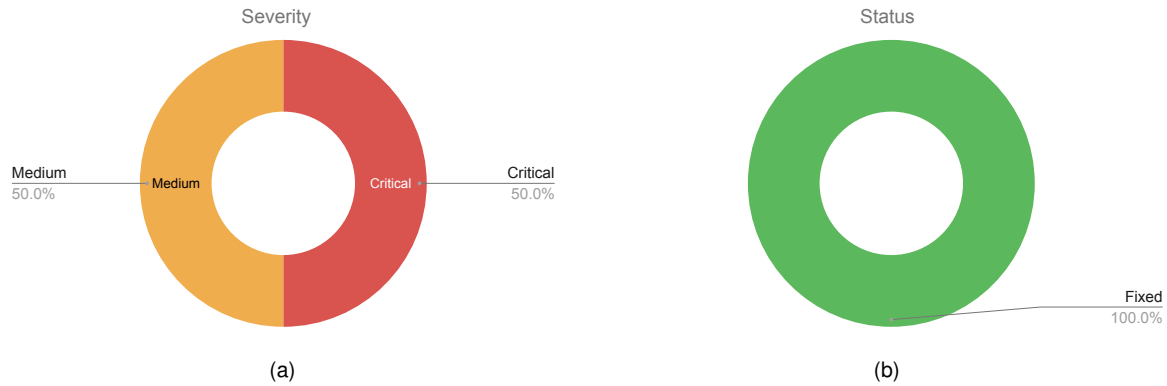

Fig. 1: Distribution of issues: Critical (1), **High** (0), **Medium** (1), **Low** (0), **Undetermined** (0), **Informational** (0), **Best Practices** (0). **Distribution of status: Fixed** (2), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	January 20, 2025
Final Report	January 20, 2025
Repository	cross-chain-deposit-module
Commit	0f924590251a212b0879b0aea1821b7e63d4ce8c
Final Commit	22e8058a238f6574b4743570c53a7358f8809838
Documentation	Docs
Documentation Assessment	High
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/core/DepositLocker.sol	719	553	76.9%	221	1493
2	src/core/DepositExecutor.sol	491	473	96.3%	164	1128
3	src/libraries/CCDMPayloadLib.sol	102	66	64.7%	17	185
4	src/libraries/CCDMFeeLib.sol	9	14	155.6%	4	27
	Total	**1321**	**1106**	**83.7%**	**406**	**2833**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect accounting in `withdrawMerkleDeposit(...)` leads to loss of funds	Critical	Fixed
2	Bridging LP tokens with skewed pool ratios can result in funds being locked	Medium	Fixed

4 System Overview

The overview of the entire CCDM system can be found in the CCDM audit report: NM-0366-FINAL_CCDM.

This section focuses exclusively on the newly added functionality of **Merkle Deposits** and **Merkle Withdrawals**. These features extend CCDM's capabilities by introducing Merkle tree-based mechanisms for aggregating and verifying deposits and withdrawals.

On the source chain, the `DepositLocker` contract facilitates deposits using the `merkleDeposit` function. Key information, such as the deposit nonce, depositor's address, and deposit amount, is hashed into a leaf node and added to the Merkle tree. Once all deposits for a market are complete, and the market is marked as ready to bridge, the `merkleBridgeSingleTokens` function can be called. This function finalizes the Merkle root, writes it into the `composeMsg`, and sends it to the destination chain using the LayerZero protocol. For markets utilizing Uniswap V2 LP tokens, the equivalent bridging function is `merkleBridgeLpTokens`.

During the audit, a market halting feature was introduced to enhance control and security. Once the system administrator halts a market, no further deposits can be made to that market. Additionally, funds in the halted market cannot be bridged to the destination chain. However, users retain the ability to withdraw their Merkle deposits from the source chain using the `merkleWithdraw` function. The halting action is irreversible.

On the destination chain, the `DepositExecutor` contract includes the `withdrawMerkleDeposit` function. To withdraw their funds, users must provide a Merkle proof and the original data used to create the leaf on the source chain. If the recipe associated with the market has been successfully executed on the destination chain, users will receive the corresponding receipt tokens. If the recipe was not executed, users can withdraw the tokens they originally deposited.

These updates improve the CCDM system by enabling efficient aggregation and verification of deposits while adding safeguards and flexibility for deposit and withdrawal processes.

The primary objective was to support bridging a significantly larger number of users compared to the previous design.

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] Incorrect accounting in `withdrawMerkleDeposit(...)` leads to loss of funds

File(s): src/core/DepositExecutor.sol

Description: The `withdrawMerkleDeposit(...)` function can be called on the destination chain to withdraw the funds deposited using the `merkleDeposit(...)` function in the `DepositLocker` contract on the source chain. It transfers the receipt token and dust amount if the deposit recipe was executed. Otherwise, it transfers the original tokens by calculating the user's shares. There's an issue in `withdrawMerkleDeposit(...)` in a scenario where the recipe hasn't been executed. The problem is that the calculation of the `amountOwed` in the `withdrawMerkleDeposit(...)` should be done using the `totalMerkleTreeSourceAmountLeftToWithdraw` and not using `totalMerkleTreeAmountDepositedOnSource`. In the current implementation, the later users withdraw, the fewer tokens they will get, leading to incorrect calculation of shares.

```
function withdrawMerkleDeposit(
    address _weirollWallet,
    uint256 _merkleDepositNonce,
    uint256 _amountDepositedOnSource,
    bytes32[] calldata _merkleProof
)
    external
    nonReentrant
{
    // Instantiate Weiroll Wallet from the address
    WeirollWallet weirollWallet = WeirollWallet(payable(_weirollWallet));
    // ...
    // Get the campaign details for the source market
    DepositCampaign storage campaign = sourceMarketHashToDepositCampaign[weirollWallet.marketHash()];
    // Get the accounting ledger for this Weiroll Wallet (amount arg is repurposed as the CCDM Nonce on destination)
    WeirollWalletAccounting storage walletAccounting = campaign.weirollWalletToAccounting[_weirollWallet];
    // ...
    // Generate the leaf for this depositor to verify the proof against the root
    bytes32 depositLeaf = keccak256(abi.encodePacked(_merkleDepositNonce, msg.sender, _amountDepositedOnSource));
    // ...
    // Process the withdrawal
    // Get the original amount deposited on source
    uint256 totalMerkleTreeAmountDepositedOnSource = walletAccounting.totalMerkleTreeAmountDepositedOnSource;
    // Get the amount left to withdraw from merkle tree
    uint256 totalMerkleTreeSourceAmountLeftToWithdraw = walletAccounting.totalMerkleTreeSourceAmountLeftToWithdraw;
    if (weirollWallet.executed()) {
        // ...
    } else {
        // ...
        // If deposit recipe hasn't been executed, return the depositor's share of the input tokens
        for (uint256 i = 0; i < campaign.inputTokens.length; ++i) {
            // Get the amount of this input token deposited by the depositor
            ERC20 inputToken = campaign.inputTokens[i];
            uint256 totalAmountDepositedOnDest = walletAccounting.tokenToTotalAmountDepositedOnDest[inputToken];

            // Make sure that the depositor can withdraw all campaign's input tokens atomically to avoid race conditions
            ↪  with recipe execution
            require(totalAmountDepositedOnDest > 0, WaitingToReceiveAllTokens());

            // @audit-issue totalAmountDepositedOnDest will be reduced proportionally to the amount withdrawn, whereas
            ↪  totalMerkleTreeAmountDepositedOnSource remains the same
            // Calculate the tokens owed to the depositor
            uint256 amountOwed = (totalAmountDepositedOnDest * _amountDepositedOnSource) /
            ↪  totalMerkleTreeAmountDepositedOnSource;
            // Account for withdrawal
            walletAccounting.tokenToTotalAmountDepositedOnDest[inputToken] -= amountOwed;

            // Transfer the amount deposited back to the depositor
            inputToken.safeTransfer(msg.sender, amountOwed);
        }
    }
    // ...
}
```

Consider the following scenario:

- There are five deposits on the source chain for 20 LP tokens each. Thus, a total of 100 LP tokens;
- While bridging, 100 LP tokens were equal to 200 `token0` and 200 `token1`;
- Now, in case the recipe isn't executed and it's a Merkle tree withdrawal:
  - The first user to call `withdrawMerkleDeposit(...)` will get 40 `token0` and 40 `token1` as `totalAmountDepositedOnDest` will be 200, `_amountDepositedOnSource` will be 20, and `totalMerkleTreeAmountDepositedOnSource` will be 100.
  - The second user to withdraw will get 32 `token0` and 32 `token1` as `totalAmountDepositedOnDest` will be 160, `_amountDepositedOnSource` will be 20, and `totalMerkleTreeAmountDepositedOnSource` will be 100. It is incorrect as the second one to call should also get 40 `token0` and 40 `token1` as all the depositors have the same shares.

Recommendation(s): Consider replacing `totalMerkleTreeAmountDepositedOnSource` with `totalMerkleTreeSourceAmountLeftToWithdraw` in the calculation of `amountOwed` in the `withdrawMerkleDeposit(...)` for the case in which the recipe isn't executed.

Status: Fixed.

Update from the client: `af45906`

6.2 [Medium] Bridging LP tokens with skewed pool ratios can result in funds being locked

File(s): `src/core/DepositLocker.sol`

Description: The campaign owner has to call `merkleBridgeLpTokens(...)` to bridge LP tokens. This will remove liquidity from the Uniswap V2 pair. The issue is that there's no check to ensure that one (or both) of the token's quantity received after removing liquidity is non-zero. If one of the tokens's quantity is zero and for the other, it's non-zero, then the bridged token would be permanently stuck in the `DepositExecutor`. This happens since the campaign owner cannot call `executeDepositRecipes(...)` for that market as it further calls `_transferInputTokensToWeirollWallet(...)`, which reverts if the token amount is zero. Nor can depositors call `withdrawMerkleDeposit(...)` as it reverts if one of the token amounts is zero for the Weiroll wallet and a withdrawal recipe isn't executed.

Recommendation(s): Consider adding a check in `merkleBridgeLpTokens(...)` to make sure that the amount of tokens received after removing liquidity is not zero.

Status: Fixed.

Update from the client: `2ce40c1`

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 CCDM documentation

The documentation for the CCDM contracts was provided through their official docs as well as detailed code comments. This documentation provided a high-level overview of the protocol and details of its implementation. Moreover, the CCDM 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

Remarks about the CCDM contract's test suite

The test suite of the CCDM contracts covers all major flows of the protocol. However, upon closer review, it was observed that the suite lacks the necessary depth to validate certain edge cases and specific state changes. This limitation makes the test suite prone to errors, such as off-by-one issues, which can compromise the reliability of the protocol in critical scenarios.
In particular, the absence of targeted assertions to verify state transitions after each operation reduces the effectiveness of the tests in identifying subtle bugs. Enhancing the granularity and precision of the assertions could significantly improve the suite's resilience.
To address these gaps, incorporating mutation testing could be an effective approach. Mutation testing introduces small changes to the contract code to verify whether the test suite is capable of detecting them. This method can highlight untested or weakly tested paths in the code, providing a clear roadmap for improving test coverage.

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

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.