Security Review Report
NM-0541 chukkerCoin

(May 15, 2025)

Contents

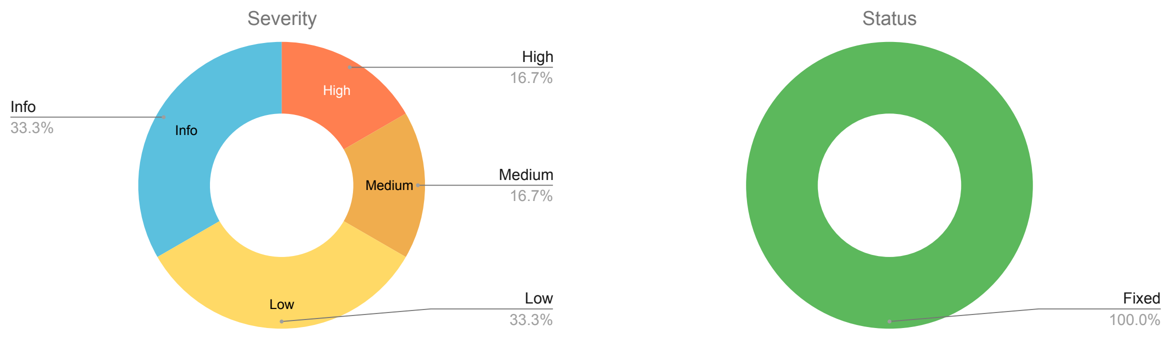
1 Executive Summary

This document presents the security review performed by Nethermind Security for chukkerCoin contracts. The audit review focused on chukkerCoin protocol which consists of two main components, the chukkerCoin token and chukkerCoinTokenSale.

The audit comprises 416 lines of solidity code. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

Along this document, we report 6 points of attention, where they are classified as one `High`, one `Medium`, two `Low`, and two `Informational`. The issues are summarized in Fig. 1. Following feedback by Nethermind Security, all issues have been fixed.

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



(a) (b)

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

Summary of the Audit

Audit Type	Security Review
Initial Report	May 7, 2025
Final Report	May 15, 2025
Repositories	chukkerCoin
Initial Commit	a361810
Final Commit	fa1dca8
Documentation	Provided in the code's natspec
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

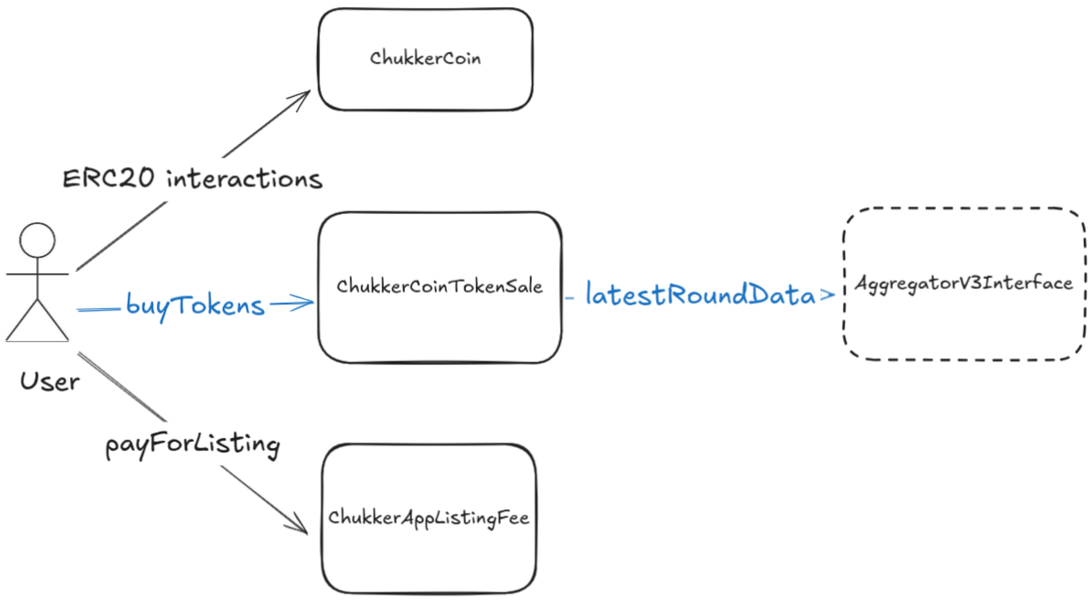
	Contract	LoC	Comments	Ratio	Blank	Total
1	interfaces/AggregatorV3Interface.sol	14	1	7.1%	5	20
2	tokensale/chukkerCoinTokenSale.sol	181	68	37.6%	57	306
3	listingfee/chukkerAppListingFee.sol	47	35	74.5%	18	100
4	token/chukkerCoin.sol	174	68	39.1%	59	301
	Total	**416**	**172**	**41.3%**	**139**	**727**

3 Summary of Issues

	Finding	Severity	Update
1	Slippage protection mechanism is ineffective	High	Fixed
2	Unsold Cc tokens are permanently locked if token sale ends early	Medium	Fixed
3	Price deviation check is unsafe due to rounding truncation	Low	Fixed
4	Stale price feeds are not fully validated	Low	Fixed
5	Unused `_burn` function	Info	Fixed
6	`answeredInRound` is deprecated	Info	Fixed

4 System Overview

The Chukker App consists of three core on-chain components: the `chukkerCoin` ERC-20 token, the `chukkerCoinTokenSale` contract that handles its public sale with Chainlink price feeds, and `ChukkerAppListingFee` that collects fees for creating listings on the ChukkerApp.



`chukkerCoin` (cC) Token

`chukkerCoin` is a gas-efficient, fully ERC-20-compliant token with built-in pausable and minting controls:

- On construction, the owner receives the full initial supply of 50 Million `cC`.
- The owner may pause and then restart all `transfer` and `transferFrom` calls to halt token movement in emergencies.
- Tokens can be minted by the owner until `finishMinting()` is called, which is at 50million `cC` at which point no further minting is allowed, and this will be the max supply.

`chukkerCoinTokenSale` Contract

This contract sells up to $30{,}000{,}000$ `cC` at a fixed price of $1 USD per token, with built-in price-feed and sale controls:

- Fetches real-time ETH/USD rates via a primary Chainlink oracle, with a fallback oracle if the primary fails.
- Requires a purchase of at least 500 `cC` ($500 USD) per call to `buyTokens`.
- If the remaining token inventory is less than the user's desired amount, the contract sells all remaining tokens and refunds any excess ETH.
- Owner can pause and restart the sale to stop new purchases; also supports emergency ETH withdrawal.
- Enforces a maximum sale cap of 30 million tokens and guards against 10% price deviations between on-chain updates.

`ChukkerAppListingFee` Contract

The `ChukkerAppListingFee` collects fees for creating listings on the ChukkerApp and allows a user to pay the listing fee.

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 [High] Slippage protection mechanism is ineffective

File(s): chukkerCoinTokenSale

Description: The chukkerCoinTokenSale::buyTokens function allows users to purchase tokens during the sale. As per the current implementation, the function appears to attempt enforcing a 1% slippage tolerance on the number of tokens issued:

```
function buyTokens() external payable nonReentrant whenNotPaused {
    // --SNIP
    // Get price and calculate token amount
    uint256 ethUsdPrice = getLatestETHPrice();
    uint256 ethValue = msg.value;

    // Calculate USD value and ensure minimum purchase
    uint256 usdValue;
    unchecked {
        usdValue = (ethValue * ethUsdPrice) / PRECISION;
    }

    if (usdValue < MIN_PURCHASE_AMOUNT) revert MinimumPurchaseRequired();

    // Calculate tokens with slippage protection
    uint256 tokenAmount;
    unchecked {
        tokenAmount = (ethValue * ethUsdPrice) / PRICE_PER_TOKEN;
    }

    uint256 minAcceptableTokens;
    unchecked {
        minAcceptableTokens = (ethValue * ethUsdPrice * (100 - SLIPPAGE_TOLERANCE)) / (PRICE_PER_TOKEN * 100);
    }

    if (tokenAmount < minAcceptableTokens) revert SlippageExceeded();
}
```

However, the current implementation does not provide real slippage protection. Both tokenAmount and minAcceptableTokens are derived from the same inputs— ethValue and ethUsdPrice. Consequently, tokenAmount will always equal or exceed minAcceptableTokens, rendering the slippage check ineffective. This means that users will receive the computed tokenAmount regardless of any actual deviation in the output they anticipated due to slippage.

Recommendation(s): Consider accepting the expected output amount from the user and assert 1% slippage protection against it.

Status: Fixed.

Update from the client: Fixed in commit e642880

6.2 [Medium] Unsold Cc tokens are permanently locked if token sale ends early

File(s): chukkerCoinTokenSale

Description: The chukkerCoinTokenSale contract is pre-funded with 30 million chukkerCoin tokens at deployment for sale to users. The contract allows the owner to terminate the sale early using the endTokenSale function:

```
function endTokenSale() external onlyOwner {
    tokenSaleEnded = true;
    emit TokenSaleEnded(totalTokensSold);
}
```

This termination is irreversible, and the function does not include any mechanism to recover unsold tokens. As a result, if the token sale is ended before all tokens are sold, the remaining chukkerCoin tokens will be permanently locked within the contract, making them inaccessible and unusable.

Recommendation(s): Consider withdrawing any leftover chukkerCoin tokens when ending token sale.

Status: Fixed.

Update from the client: Fixed in commit 5ab5db3

6.3 [Low] Price deviation check is unsafe due to rounding truncation

File(s): chukkerCoinTokenSale

Description: The getLatestETHPrice function is used within buyTokens to fetch the latest ETH/USD price and includes a safeguard to reject prices with deviations greater than 10%:

```
function getLatestETHPrice() public returns (uint256) {
    // --SNIP
    if (lastPrice > 0) {
        uint256 deviation;
        unchecked {
            deviation = currentPrice > lastPrice
                ? ((currentPrice - lastPrice) * 100) / lastPrice
                : ((lastPrice - currentPrice) * 100) / lastPrice;
        }

        if (deviation > MAX_PRICE_DEVIATION) {
            revert PriceDeviationExceeded();
        }
    }
}
```

However, this approach suffers from rounding truncation when computing the deviation. For example, if currentPrice is 1109e18 ($1109) and lastPrice is 1000e18 ($1000), the actual deviation is 10.1%, but due to integer division, the calculated result is truncated to 10%, allowing the transaction to proceed when it should be reverted. This leads to acceptance of price deviations slightly over the allowed threshold.

Recommendation(s): Consider replacing the division-based check with a cross-multiplication approach, which retains full precision, a possible approach would be:

```
uint256 diff = currentPrice > lastPrice
    ? currentPrice - lastPrice
    : lastPrice - currentPrice;

if (diff * 100 > lastPrice * MAX_PRICE_DEVIATION) {
    revert PriceDeviationExceeded();
}
```

Status: Fixed.

Update from the client: Fixed in commit 49db7b5

6.4 [Low] Stale price feeds are not fully validated

File(s): chukkerCoinTokenSale

Description: The getLatestETHPrice function retrieves the ETH/USD price from a Chainlink price feed and performs several validity checks on the response to ensure it is not stale:

```
function getLatestETHPrice() public returns (uint256) {
    AggregatorV3Interface priceFeed = useFallbackOracle ? SECONDARY_PRICE_FEED : ETH_USD_PRICE_FEED;
    try priceFeed.latestRoundData() returns (
        uint80 roundId, int256 price, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound
    ) {
        // Validate price data
        if (roundId == 0 || startedAt == 0 || updatedAt < startedAt || answeredInRound < roundId) {
            revert InvalidPriceData();
        }
    }
    // --SNIP
}
```

While the function includes several validation steps, it does not check whether the updatedAt timestamp is too old. As a result, if the price feed becomes stale—e.g., due to downtime or delays—the function may return outdated prices, which will lead to incorrect token pricing.

Recommendation(s): Consider implementing an additional check to ensure that the updatedAt timestamp is within an acceptable freshness window (e.g., not older than 1 hour).

Status: Fixed.

Update from the client: Fixed in commit fb4cf31

6.5 [Info] Unused _burn function

File(s): chukkerCoin

Description: The chukkerCoin token contract defines an internal _burn function intended to allow token burning. However, this function is currently unused, as there are no public or external functions within the contract that invoke it. As a result, the _burn function is effectively dead code and it is unreachable.

Recommendation(s): Consider revisiting the use of _burn.

Status: Fixed.

Update from the client: Fixed in commit fa1dca8

6.6 [Info] answeredInRound is deprecated

File(s): chukkerCoinTokenSale

Description: The getLatestETHPrice function retrieves the ETH/USD price from a Chainlink price feed and performs several validity checks to ensure the data is not stale or invalid:

```
function getLatestETHPrice() public returns (uint256) {
    AggregatorV3Interface priceFeed = useFallbackOracle ? SECONDARY_PRICE_FEED : ETH_USD_PRICE_FEED;
    try priceFeed.latestRoundData() returns (
        uint80 roundId, int256 price, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound
    ) {
        // Validate price data
        if (roundId == 0 || startedAt == 0 || updatedAt < startedAt || answeredInRound < roundId) {
            revert InvalidPriceData();
        }
    }
    // --SNIP
}
```

The check answeredInRound < roundId is no longer meaningful, as answeredInRound has been deprecated. According to the Chainlink documentation, answers are now computed and returned within the same round, making this comparison redundant.

Recommendation(s): Consider removing the answeredInRound < roundId check to simplify the validation logic and avoid reliance on deprecated fields.

Status: Fixed.

Update from the client: Fixed in commit fb4cf31

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 chukkerCoin documentation

The `chukkerCoin` team has provided a comprehensive walkthrough of the project in the kick-off call, and the code natspec includes a detailed explanation of the intended functionalities. Moreover, the 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 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 43 files with Solc 0.8.20
[] Solc 0.8.20 finished in 1.91s
Compiler run successful with warnings:
Warning (2018): Function state mutability can be restricted to view
   --> test/token/ChukkerCoin.t.sol:159:5:
    |
159 |     function testDomainSeparator() public {
    |     ^ (Relevant source part starts here and spans across multiple lines).
```

8.2 Tests Output

```
> forge test
Ran 14 tests for test/token_sale/ChukkerCoinTokenSale.t.sol:ChukkerCoinTokenSaleTest
[PASS] testConcurrentPurchases() (gas: 281726)
[PASS] testEmergencyWithdraw() (gas: 148634)
[PASS] testEndTokenSale() (gas: 54823)
[PASS] testFallbackPriceFeed() (gas: 160771)
[PASS] testInitialSetup() (gas: 26890)
[PASS] testMinAcceptableTokens() (gas: 15133)
[PASS] testPartialPurchase() (gas: 166765)
[PASS] testPauseAndUnpause() (gas: 35720)
[PASS] testPriceFeedIntegration() (gas: 139541)
[PASS] testSlippageProtection() (gas: 152521)
[PASS] testTogglePriceFeed() (gas: 38479)
[PASS] testTokenPurchase() (gas: 139563)
[PASS] test_RevertWhen_EmergencyWithdrawNoBalance() (gas: 13024)
[PASS] test_RevertWhen_PurchaseAmountTooSmall() (gas: 64231)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 5.39ms (3.81ms CPU time)

Ran 14 tests for test/listing_fee/ChukkerAppListingFee.t.sol:ChukkerAppListingFeeTest
[PASS] testConstructor() (gas: 21480)
[PASS] testConstructorRevertZeroAddress() (gas: 84179)
[PASS] testConstructorRevertZeroFee() (gas: 84341)
[PASS] testGetListingFee() (gas: 12543)
[PASS] testPayForListing() (gas: 69942)
[PASS] testPayForListingInsufficientAllowance() (gas: 26331)
[PASS] testSetListingFee() (gas: 21560)
[PASS] testSetListingFeeRevertZero() (gas: 10614)
[PASS] testSetListingFeeUnauthorized() (gas: 14066)
[PASS] testWithdrawTokens() (gas: 85017)
[PASS] testWithdrawTokensExceedingBalance() (gas: 72831)
[PASS] testWithdrawTokensUnauthorized() (gas: 14332)
[PASS] testWithdrawTokensZeroAddress() (gas: 68653)
[PASS] testWithdrawTokensZeroAmount() (gas: 17993)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 26.55ms (2.62ms CPU time)

Ran 4 tests for test/token/ChukkerCoin.fuzz.t.sol:ChukkerCoinFuzzTest
[PASS] testFuzz_ApproveAndTransferFrom(uint256,uint256) (runs: 256, : 82603, ~: 84434)
[PASS] testFuzz_MultipleTransfers(uint256[]) (runs: 256, : 85419, ~: 84805)
[PASS] testFuzz_Permit(uint256,uint256,uint256) (runs: 256, : 122455, ~: 122244)
[PASS] testFuzz_Transfer(uint256) (runs: 256, : 54802, ~: 55255)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 198.87ms (325.47ms CPU time)
```

```
Ran 14 tests for test/token/ChukkerCoin.t.sol:ChukkerCoinTest
[PASS] testAllowance() (gas: 100536)
[PASS] testApproveAndTransferFrom() (gas: 75667)
[PASS] testCannotMintAfterFinishMinting() (gas: 39358)
[PASS] testDomainSeparator() (gas: 10495)
[PASS] testInitialSupply() (gas: 16203)
[PASS] testMinting() (gas: 73009)
[PASS] testMintingRestrictions() (gas: 51093)
[PASS] testPause() (gas: 56033)
[PASS] testPermit() (gas: 110656)
[PASS] testPermitExpired() (gas: 49207)
[PASS] testTokenMetadata() (gas: 13579)
[PASS] testTransfer() (gas: 47069)
[PASS] test_RevertWhen_PauseUnauthorized() (gas: 13089)
[PASS] test_RevertWhen_TransferInsufficientBalance() (gas: 18854)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 1.82s (8.23ms CPU time)

Ran 10 tests for test/token_sale/ChukkerCoinTokenSale.fuzz.t.sol:ChukkerCoinTokenSaleFuzzTest
[PASS] testFuzz_CalculateMinAcceptableTokens(uint256,int256) (runs: 256, : 18127, ~: 18181)
[PASS] testFuzz_CalculateTokenAmount(uint256,int256) (runs: 256, : 16417, ~: 16455)
[PASS] testFuzz_FallbackPriceFeed(uint256) (runs: 256, : 168833, ~: 168849)
[PASS] testFuzz_MaxPurchase(uint256) (runs: 256, : 141115, ~: 141143)
[PASS] testFuzz_MinPurchase(uint256) (runs: 256, : 89977, ~: 68893)
[PASS] testFuzz_MultipleUsers(address[]) (runs: 256, : 207987, ~: 211100)
[PASS] testFuzz_PriceFeedIntegration(int256) (runs: 256, : 145799, ~: 145791)
[PASS] testFuzz_PriceManipulation(int256) (runs: 256, : 144515, ~: 144506)
[PASS] testFuzz_SlippageProtection(uint256) (runs: 256, : 191036, ~: 191060)
[PASS] testFuzz_TokenPurchase(uint256) (runs: 256, : 144731, ~: 144771)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 1.82s (2.09s CPU time)

Ran 5 test suites in 1.83s (3.87s CPU time): 56 tests passed, 0 failed, 0 skipped (56 total tests)
```

8.3 Automated Tools

8.3.1 AuditAgent

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

9 About Nethermind

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

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

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

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

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

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

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

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

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;

- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;

- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer