Security Review Report
NM-0382 WORLD NFC-ID

NETHERMIND SECURITY

(Nov 14, 2024)

Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the WORLD NFC-ID contracts. The codebase is an evolution of the codebase audited in the report NM-0302-WORLDCOIN-GRANTS4 contracts. This audit focused on two contracts: src/NFC_ID/NFC_ID.sol and src/GatedMulticall3/GatedMulticall3.sol comprising 281 lines of Solidity code, and it was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract. The codebase has high quality.

Along this document, we report six points of attention, where one is classified as `Info` and five are classified as `Best Practices`. The issues are summarized in Fig. 1.

The issues relate to best practices that enhance code consistency and pose no security threats. For example, missing zero-address checks, unnecessary `payable` keywords, and omitted event emissions do not compromise contract security but can improve reliability and maintainability when addressed.

Out of the six issues identified, three were `fixed`: First, the issue with events not being emitted in the constructor was resolved, ensuring that storage variable updates trigger events as expected. Second, a zero-address check was added to the `addAuthorizedCaller` function, aligning it with the rest of the codebase's standards for address validation. Third, an unchecked conversion on the `_amount` parameter in the `claim(...)` function was addressed by implementing a check to prevent overflow, thus improving reliability.

The remaining three issues were `acknowledged`. Centralization risks were accepted. Additionally, functions that don't expect native tokens to be transferred were left marked as `payable` to maintain compatibility with the `Allowance Module` interface, avoiding interface modifications. Finally, the lack of enforcement linking `_nullifierHash` and `_recipient` was accepted as a manageable risk, given that only trusted callers can access the function.

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

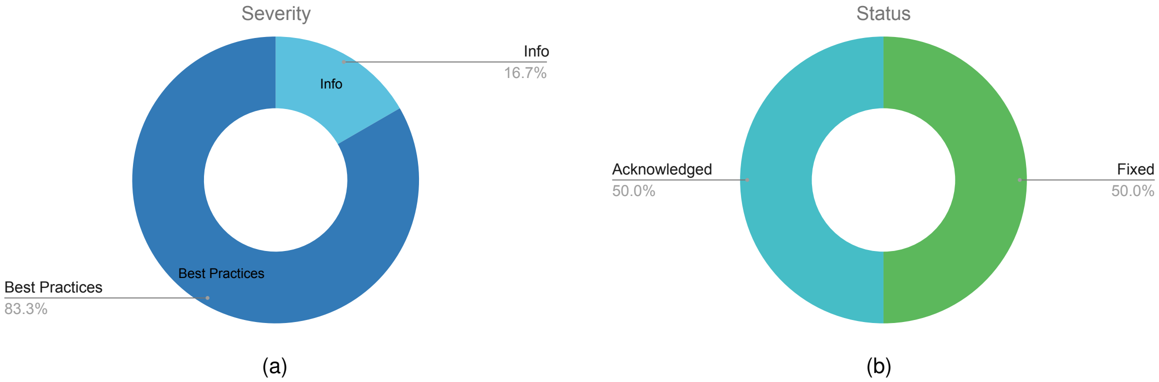

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (1), **Best Practices** (5). **Distribution of status: Fixed** (3), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Nov 15, 2024
Response from Client	Regular responses during audit engagement
Final Report	Nov 18, 2024
Repository	World NFC-ID
Commit (Audit)	8672196e6b647c0069fc1c95eb392455b0ddbe61
Commit (Final)	d82b025501f3d4d90899b6ed20ca1e1676efe70e
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/GatedMulticall3/GatedMulticall3.sol	186	69	37.1%	41	296
2	src/NFC_ID/NFC_ID.sol	95	40	42.1%	34	169
	Total	**281**	**109**	**38.8%**	**75**	**465**

3 Summary of Issues

	Finding	Severity	Update
1	Centralization Risks	Info	Acknowledged
2	Functions that don't expect native tokens to be transferred shouldn't be payable	Best Practices	Acknowledged
3	Events not emitted in the constructor	Best Practices	Fixed
4	Lack of zero address check on `GatedMulticall3::addAuthorizedCaller` function	Best Practices	Fixed
5	Unchecked conversion on `_amount` in the function `claim(...)`	Best Practices	Fixed
6	Lack of enforcement linking `_nullifierHash` and `_recipient`	Best Practices	Acknowledged

4 System Overview

The following section provides an overview of the system architecture and key contracts involved in the current Worldcoin Grants contracts (Grants4) version, which replaces the previous contracts starting from the 39th grant.

4.1 Contracts overview

The Worldcoin Grants4 system is primarily composed of the following contracts:

- **RecurringGrantDrop (new)**: This contract handles the distribution of grant tokens (WLD) for grants with a `grantId` of 39 or higher. The contract allows users to claim monthly WLD drops once the Orb verifies them. The contract supports the new feature where authorized relayers can claim drops on behalf of users immediately after their verification.
- **WLDGrant**: This contract holds the configuration of the Grants contracts. It is used to query the currently active grants and their token amounts. The `checkValidity(...)` function can be used to determine if a particular `grantId` is claimable.
- **Grants4FirstBatch (replaced)**: Previously, this contract allowed authorized Worldcoin relayers to call the `batch(...)` function to claim drops on behalf of users. This action marked the `nullifierHash` as used in the `RecurringGrantDrop` contract to prevent users from claiming the same grant twice. This was replaced by the `NFC_ID` contract.
- **GatedMulticall3 (new)**: This contract is a modified version of the well-known `Multicall3`, adding an authorization layer. Only authorized `msg.sender` accounts can interact with it, ensuring secure execution of multiple contract calls in a single transaction.
- **NFC_ID (new)**: This contract replaces the `Grants4FirstBatch`. It allows authorized relayers to claim `WLD` tokens for individual users. Unlike its predecessor, `NFC_ID` does not handle batch claims but instead stores the `nullifierHash` directly within the contract, eliminating the need to call the `RecurringGrantDrop` contract for nullifier validation.

The grant-claiming process in the Worldcoin Grants4 system remains largely similar, depending on whether the grant is part of the new grant cycle (Grants 39-88) or involves reservations made in the previous system. The following section outlines the updated grant-claiming process.

4.2 Grant claiming process

For the **new grants cycle**, starting from Grant 39 and continuing through to Grant 88, users are entitled to claim WLD token drops every month. the new system introduces a decreasing grant amount each month. However, the initial month's token amount starts from a higher base than in the previous system, providing a larger initial drop. The specific amounts are predetermined and stored in the `grantAmountsList` array in the `WLDGrant` contract. Whenever the user or authorized relayer claims a grant, the `RecurringGrantDrop` contract queries the `WLDGrant` contract to determine the exact token amount to be distributed for that particular month.

At any given time, two grants are active: the current month's grant and the previous month's grant. This allows users who missed claiming their tokens in the current month to still claim them in the following month.

For **old reservations** (Grants 21-38), users can still claim their previously reserved tokens by interacting with the old instance of the `RecurringGrantDrop` contract.

4.3 Immediate post-verification grant claims

One of the enhancements in the `Worldcoin Grants4` protocol is the introduction of authorized relayers who can claim `WLD` token drops on behalf of users immediately after they are verified by the Orb. This feature improves the user experience by eliminating delays that existed in the previous system.

Previously, users had to wait until their identity was propagated across all relevant Layer 2 (L2) networks before claiming their tokens, causing a delay between verification and the ability to claim. In the new system, authorized Worldcoin relayers interact with the `NFC_ID` contract to claim tokens for users individually.

When a claim is made, the `NFC_ID` contract validates the `nullifierHash` directly. If the `nullifierHash` has not been used, it is marked as used within `NFC_ID`, and the `WLD` tokens are transferred to the user, completing the claim. If the `nullifierHash` has already been set, the claim is skipped, preventing duplicate claims.

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 [Info] Centralization Risks

File(s): src/NFC_ID/NFC_ID.sol

Description: The contract allows the owner to update the address of the WLD token, the HOLDER, and the ALLOWANCE_MODULE. Arbitrary changes in critical configuration contracts may open the door to malicious modifications compromising contract security. The functions setWldToken(...), setHolder(...), and setAllowanceModule(...) are reproduced below.

```
function setWldToken(address _wldToken) external onlyOwner {
    if (_wldToken == address(0)) {
        revert ZeroAddress();
    }
    WLD_TOKEN = _wldToken;
    emit WldTokenSet(_wldToken);
}
```

```
function setHolder(address _holder) external onlyOwner {
    if (_holder == address(0)) {
        revert ZeroAddress();
    }
    HOLDER = GnosisSafe(_holder);
    emit HolderSet(_holder);
}
```

```
function setAllowanceModule(address _allowanceModuleAddress) external onlyOwner {
    if (_allowanceModuleAddress == address(0)) {
        revert ZeroAddress();
    }
    ALLOWANCE_MODULE = AllowanceModule(_allowanceModuleAddress);
    emit AllowanceModuleSet(_allowanceModuleAddress);
}
```

Recommendation(s): No action is required.

Status: Acknowledged

Update from client: While it is true that it is questionable that we will ever update the token address, etc. The owner will be a trusted multisig, and we accept these risks.

6.2 [Best Practice] Functions that don't expect native tokens to be transferred shouldn't be payable

File(s): src/GatedMulticall3/GatedMulticall3.sol

Description: The aggregate(...), tryAggregate(...), tryBlockAndAggregate(...), blockAndAggregate(...), and aggregate3(...) function doesn't expect any native tokens to be transferred unlike aggregate3Value(...) so they shouldn't be payable. Any native token sent will be locked in the contract.

Recommendation(s): Consider removing the payable keyword in the functions that don't expect native tokens to be transferred.

Status: Acknowledged

Update from client: We won't fix it. The Allowance Module interface expects a payable address, so to avoid changes to the interface, we keep marking it as payable.

Update from Nethermind: We agree with this decision.

6.3 [Best Practices] Events not emitted in the constructor

File(s): src/NFC_ID/NFC_ID.sol

Description: When ALLOWANCE_MODULE, WLD_TOKEN or HOLDER is changed using setAllowanceModule(...), setWldToken(...) or setHolder(...) function respectively, event AllowanceModuleSet, WldTokenSet or HolderSet is emitted but when these variables are initially set in the constructor, no events are emitted.

Recommendation(s): Emit appropriate events in the constructor as the storage variables are also updated in the constructor.

Status: Fixed

Update from client: Added in d82b025501f3d4d90899b6ed20ca1e1676efe70e.

6.4 [Best Practices] Lack of zero address check on GatedMulticall3::addAuthorizedCaller function

File(s): src/GatedMulticall3/GatedMulticall3.sol

Description: All the setter functions inside the codebase have a check that prevents the owner from giving any roles to address(0x0), except for the function GatedMulticall3::addAuthorizedCaller(...). The function is reproduced below.

```
function addAuthorizedCaller(address _caller) public onlyOwner {
    ////////////////////////////////////////////////
    //@audit: lack of zero address check
    ////////////////////////////////////////////////
    authorizedCallers[_caller] = true;
    emit AuthorizedCallerAdded(_caller);
}
```

Recommendation(s): Consider adding a zero address check in this function to maintain consistency throughout the codebase. However, there is no harm in keeping the code as it is.

Status: Fixed

Update from client: Fixed in d82b025501f3d4d90899b6ed20ca1e1676efe70e.

6.5 [Best Practices] Unchecked conversion on _amount in the function claim(...)

File(s): src/NFC_ID/NFC_ID.sol

Description: The function claim(...) converts the variable _amount from uint256 to uint96 but there is no check to ensure that _amount actually fits within a uint96.

```
function claim( uint256 _nullifierHash, address _recipient, uint256 _amount ) external {
    ...
    /////////////////////////////////////////////////////////
    // @audit: Check if "_amount" fits in a uint96 before
    //         making the conversion.
    /////////////////////////////////////////////////////////
    AllowanceModule(ALLOWANCE_MODULE).executeAllowanceTransfer(
        HOLDER, WLD_TOKEN, payable(_recipient), uint96(_amount)
    );

    emit NFCIDGrantClaimed(_recipient);
}
```

Recommendation(s): Add a check to ensure that _amount always fits a uint96 before making the conversion.

Status: Fixed

Update from client: Fixed in d82b025501f3d4d90899b6ed20ca1e1676efe70e.

6.6 [Best practices] Lack of enforcement linking `_nullifierHash` and `_recipient`

File(s): `src/NFC_ID/NFC_ID.sol`

Description: The function `NFC_ID::claim(...)` does not enforce any relationship between the `_nullifierHash` and the `_recipient` parameters. As long as the caller provides a valid `_nullifierHash`, they can set their own address as the `_recipient`, claiming the tokens meant for other users.

Since this function can only be called by a whitelisted `allowedCaller`, the impact is mitigated, but the security of this function can be further improved if these two parameters are tied together.

Recommendation(s): Consider adding checks that tie the `_nullifierHash` to the adequate `_recipient`.

Status: Acknowledged

Update from client: We won't fix it. The contract can only be called by a trusted caller, and we accept this risk.

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 World documentation

The **World** team has provided documentation about their protocol in the form of in-line comments within the code.

8 Test Suite Evaluation

```
forge test
[] Compiling...
[] Compiling 44 files with 0.8.19
[] Solc 0.8.19 finished in 2.96s
Compiler run successful with warnings:

Ran 6 tests for src/test/WLDGrantPreGrant4_new.t.sol:WLDGrantTest
[PASS] testFuzz_checkReservationValidityReverts(uint256) (runs: 256, : 11029, ~: 11029)
[PASS] testFuzz_checkValidityReverts(uint256) (runs: 256, : 9954, ~: 9955)
[PASS] test_checkReservation_grant37() (gas: 11367)
[PASS] test_checkReservation_grant38() (gas: 11249)
[PASS] test_checkValidity_grant38() (gas: 10110)
[PASS] test_checkValidity_revertsGreaterThanGrant38() (gas: 10793)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 85.60ms (138.72ms CPU time)

Ran 17 tests for src/test/WLDGrant.t.sol:WLDGrantTest
[PASS] testFuzz_getAmount_RevertsIfOutsideBounds(uint256) (runs: 256, : 9139, ~: 9141)
[PASS] test_checkValidity_August2024_grant39() (gas: 10715)
[PASS] test_checkValidity_February2026_grant56() (gas: 11172)
[PASS] test_checkValidity_Feburary2026_grant57() (gas: 11206)
[PASS] test_checkValidity_January2026_grant55() (gas: 11149)
[PASS] test_checkValidity_January2026_grant56() (gas: 11197)
[PASS] test_checkValidity_September2024_grant39() (gas: 11216)
[PASS] test_checkValidity_September2024_grant40() (gas: 11173)
[PASS] test_checkValidity_revertIfGrantIdLessThan21() (gas: 8641)
[PASS] test_checkValidity_revertIfGrantIdLessThan38ButGrant4LaunchHappened() (gas: 8961)
[PASS] test_getAmount_grant30() (gas: 8638)
[PASS] test_getAmount_grant38() (gas: 8646)
[PASS] test_getAmount_grant39() (gas: 7848)
[PASS] test_getAmount_grant40() (gas: 7870)
[PASS] test_getAmount_grant75() (gas: 7858)
[PASS] test_getAmount_grant88() (gas: 7848)
[PASS] test_getAmount_grant89() (gas: 8659)
Suite result: ok. 17 passed; 0 failed; 0 skipped; finished in 88.46ms (78.28ms CPU time)

Ran 3 tests for src/test/NFC_ID.t.sol:NFC_IDTest
[PASS] test_claim() (gas: 109425)
[PASS] test_claim_reverts_if_caller_not_allowed() (gas: 11382)
[PASS] test_claim_reverts_if_nullifier_hash_already_set() (gas: 113863)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 106.80ms (1.05ms CPU time)

Ran 12 tests for src/test/WLDGrantReservations.t.sol:WLDGrantReservationsTest
[PASS] testFuzz_checkReservationValidityReverts(uint256) (runs: 256, : 10471, ~: 11137)
[PASS] test_checkReservationValidity_grant13() (gas: 10985)
[PASS] test_checkReservationValidity_grant38() (gas: 11448)
[PASS] test_checkReservationValidity_revertsGreaterThanGrant38() (gas: 12174)
[PASS] test_checkReservationValidity_revertsLessThan13() (gas: 9513)
[PASS] test_checkReservation_grant37() (gas: 11448)
[PASS] test_checkReservation_grant38() (gas: 11376)
[PASS] test_getAmount_grant13() (gas: 5510)
[PASS] test_getAmount_grant14() (gas: 5511)
[PASS] test_getAmount_grant15() (gas: 5581)
[PASS] test_getAmount_grant30() (gas: 5536)
[PASS] test_getAmount_grant38() (gas: 5591)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 120.04ms (45.65ms CPU time)

Ran 3 tests for src/test/LaunchGrant.t.sol:MonthlyGrantTest
[PASS] testBiWeeklySwitch() (gas: 23639)
[PASS] testConsecutiveSpecialWeeks() (gas: 13644)
[PASS] testInitialLaunch2Weeks() (gas: 17927)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 149.07ms (2.36ms CPU time)
```

```
Ran 18 tests for src/test/RecurringGrantDrop.t.sol:RecurringGrantDropTest
[PASS] testFuzz_CannotAddNullifierHashBlockerIfNotManager(address) (runs: 256, : 14040, ~: 14040)
[PASS] test_CanAddNullifierHashBlockerIfManager() (gas: 35982)
[PASS] test_CanClaim39(uint256,uint256) (runs: 256, : 131105, ~: 131105)
[PASS] test_CanClaim39_2ndMonth(uint256,uint256) (runs: 256, : 131605, ~: 131605)
[PASS] test_CannotClaimAlreadyClaimedGrant_39(uint256,uint256) (runs: 256, : 135044, ~: 135044)
[PASS] test_CannotClaimAlreadyClaimedGrant_40(uint256,uint256) (runs: 256, : 135504, ~: 135504)
[PASS] test_CannotClaimFuture_22(uint256,uint256) (runs: 256, : 50161, ~: 50161)
[PASS] test_CannotClaimFuture_55(uint256,uint256) (runs: 256, : 53118, ~: 53118)
[PASS] test_CannotClaimPastGrant_21(uint256,uint256) (runs: 256, : 50243, ~: 50243)
[PASS] test_CannotClaimPastGrant_39(uint256,uint256) (runs: 256, : 53172, ~: 53172)
[PASS] test_CannotDoubleClaim(uint256,uint256) (runs: 256, : 137207, ~: 137207)
[PASS] test_CannotUpdateGrantIfNotManager(address) (runs: 256, : 2576782, ~: 2576782)
[PASS] test_UpdateGrant() (gas: 2579739)
[PASS] test_cannotClaimBelow39_21(uint256,uint256) (runs: 256, : 41286, ~: 41286)
[PASS] test_cannotClaimBelow39_38(uint256,uint256) (runs: 256, : 41341, ~: 41341)
[PASS] test_setNullifierHash_canBeCalledByAllowedBlocker() (gas: 60648)
[PASS] test_setNullifierHash_revertsIfCalledTwice() (gas: 62503)
[PASS] test_setNullifierHash_revertsIfNotAllowed(address) (runs: 256, : 11321, ~: 11321)
Suite result: ok. 18 passed; 0 failed; 0 skipped; finished in 162.13ms (1.28s CPU time)

Ran 6 test suites in 199.42ms (712.10ms CPU time): 59 tests passed, 0 failed, 0 skipped (59 total tests)
```

Remarks about World test suite

The **World** team has a comprehensive test suite that checks the main features of the codebase. This rigorous testing contributes to a well-written and secure codebase.

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.