Security Review Report NM-0291
Braavos Session Keys & Account Factory

(November 6, 2024)

Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind Security for the Braavos Account Wallet, smart account implementation designed for the Starknet ecosystem, taking advantage of the built-in account abstraction features from the Starknet network.

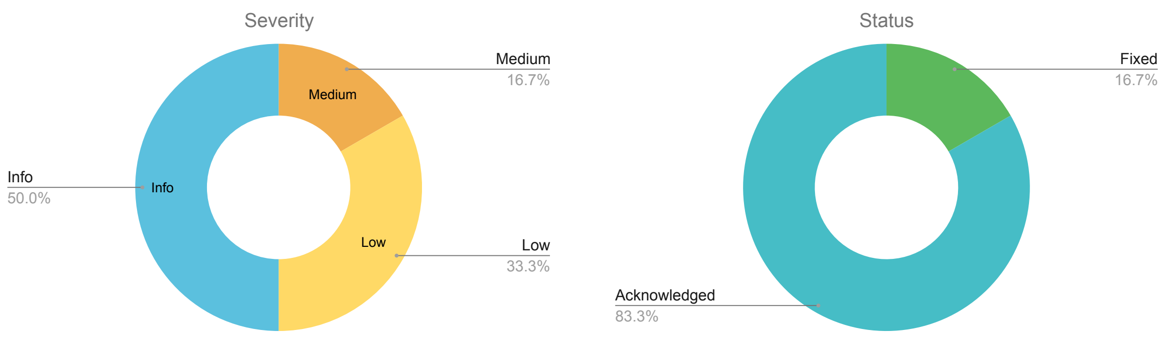
The wallet implementation has previously been audited by Nethermind, and this report presents the review of additional features that have been introduced since the last audit engagement. The newly introduced features include **factory deployment** which allows other smart wallets to deploy new Braavos wallets, **hashing changes** to utilize the new Cairo builtin hash features, and **sessions** which allow a wallet owner to give permission to another entity to execute particular transactions on the user's behalf.

The audited code comprises of 1270 lines of code written in the Cairo language, and the audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract. At the end of the engagement after all fixes were applied, the class hashes for the base account and full account implementations are listed below:

- Base class hash: `0x03d16c7a9a60b0593bd202f660a28c5d76e0403601d9ccc7e4fa253b6a70c201`
- Account class hash: `0x02c8c7e6fbcfb3e8e15a46648e8914c6aa1fc506fc1e7fb3d1e19630716174bc`

Along this document, we report six points of attention, where one is classified as `Medium`, two are classified as `Low`, and three are classified as `Informational`. 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 discusses the risk rating methodology. Section 5 details the issues. Section 6 discusses the documentation provided by the client for this audit. Section 7 presents the compilation, tests, and automated tests. Section 8 concludes the document



(a) distribution of issues according to the severity (b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (2), **Undetermined** (0), **Informational** (3), **Best Practices** (0). **(b) Distribution of status: Fixed** (1), **Acknowledged** (5), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	August 22, 2024
Final Report	November 6, 2024
Methods	Manual Review, Automated analysis
Repository	Private at time of review
Commit Hash	`2421508e3046e1ffc280a2b9e95e59034a8a77a7`
Final Commit Hash	`ec5398bd97ed5b8f8f2e53a1c2cda2a09bb39d90`
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/lib.cairo	10	0	0.0%	0	10
2	src/presets.cairo	3	0	0.0%	0	3
3	src/account.cairo	1	0	0.0%	0	1
4	src/sessions.cairo	4	0	0.0%	0	4
5	src/presets/braavos_account.cairo	553	140	25.3%	76	769
6	src/presets/braavos_base_account.cairo	138	42	30.4%	26	206
7	src/sessions/interface.cairo	83	15	18.1%	14	112
8	src/sessions/hash.cairo	97	3	3.1%	17	117
9	src/sessions/utils.cairo	57	0	0.0%	8	65
10	src/sessions/sessions.cairo	324	44	13.6%	35	403
	Total	**1270**	**244**	**19.2%**	**176**	**1690**

3 Summary of Issues

	Finding	Severity	Update
1	Session spending limit can be evaded using DAI `transferFrom`	Medium	Fixed
2	Sessions can be re-cached without signer authorization	Low	Acknowledged
3	Tokens spent amount cannot be tracked for non-standard methods	Low	Acknowledged
4	Deployment via UDC not possible when caller has non-stark signature	Info	Acknowledged
5	Hash preprocessing for SHA-256 does not follow specification	Info	Acknowledged
6	Missing length check during stark signature validation	Info	Acknowledged

4 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

5 Issues

5.1 [Medium] Session spending limit can be evaded using DAI `transferFrom`

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

Description: When executing a session transaction, token spend limits are checked through the function `_validate_spending_limits` where any call to a budgeted token address has its function selector checked. This check is done by `is_erc20_token_removal_call` which has the following logic:

```
fn is_erc20_token_removal_call(call: @Call) -> bool {
    (*call.selector == Consts::TRANSFER_CALL_SELECTOR
        || *call.selector == Consts::APPROVE_CALL_SELECTOR
        || *call.selector == Consts::INCREASE_ALLOWANCE_CALL_SELECTOR
        || *call.selector == Consts::INCREASE_ALLOWANCE_CAMEL_CALL_SELECTOR)
        && (*call.calldata).len() == 3
}
```

The function `transferFrom` is not included in this check, since it is implied that for a `transferFrom` call to succeed, it requires either an existing allowance or an approval in a prior call, which would be tracked as part of the token spend amount. This holds true for most the vast majority of ERC20 tokens including the Openzeppelin ERC20 specification, however the DAI token logic has a unique behavior in `transferFrom` which breaks this assumption. The DAI ERC20 implimentation has an optimization in the `transferFrom` function to not check the allowance if the token sender is the same as the caller address, and instead it treats it like a normal `transfer` call, as shown below:

```
@external
func transferFrom{...}(sender : felt, recipient : felt, amount : Uint256) -> (res : felt):
    alloc_locals

    # Do the transfer first
    let (local caller) = get_caller_address()
    _transfer(sender, recipient, amount)

    # Only update allowance if caller is not the sender
    # If caller is sending then it is not necessary to update allowance
    if caller != sender:
      let (allowance) = _allowances.read(sender, caller)
      # ...
    end
    return (res=1)
end
```

If a session is created where DAI is a budgeted token but the `transferFrom` function is able to be called, then the token spend checks can be evaded allowing for a session to drain all DAI on the wallet.

Recommendation(s): Consider adding a custom check to handle `transferFrom` with DAI such that the spent amount is tracked for `transferFrom` calls where the sender is the caller. For calls where the sender address is not the same as the contract account address, the spent amount does not need to be checked as the allowance of another user would be used.

Status: Fixed

Update from the client: Fixed. We now handle `transferFrom` DAI calls with `sender` set to account contract as regular `transfer` calls.

5.2 [Low] Sessions can be re-cached without signer authorization

File(s): `src/presets/braavos_account.cairo`

Description: When a session is executed for the first time, it must pass `_is_valid_signature_common` where enough signers must have provided a valid signature for the session to be executed. As an optimization to prevent signature checks on every session execution, the session validation is cached in the `validated_sessions` mapping using a key created from `(session_hash, signer_change_index)`.

```
fn _validate_session_execute(...) -> felt252 {
    //...
    // If already cached then don't check signatures
    let is_execute_session_validated = self.is_session_validated(session_hash);
    assert(
        is_execute_session_validated
            ||
            self.get_contract()._is_valid_signature_common(
                    session_hash,
                    session_execute_request.session_request_signature,
                    timestamp,
                    tx_ver
            ) == starknet::VALIDATED,
        Errors::INVALID_SIG
    );
    //...

    // If the session signatures are valid but hasn't been cached, then cache
    if (!is_execute_session_validated) {
        self.cache_session(session_hash);
        // ...
    }
    // ...
}
```

Once a session is cached, only the session owner signature needs to be validated during a session execution. The current assumption is that every time a signer is added or removed, the `signer_change_index` will increment, which invalidates all existing cached sessions. These sessions would then require the new set of wallet signers to sign for the session again. However, this assumption is can be broken in some specific scenarios. A session owner may re-use previous signatures from the wallet signers and extract a subset of the signatures to cache the session again without requiring authorization from the signers. Below we present two scenarios where a session may be cached after a `signer_change_index` by reusing the signatures:

Scenario 1:

- Wallet has a multisig threshold of 2, with two hardware signers ;
- Both hardware signers provide signature to authorize a session ;
- The session is executed and now cached ;
- A third hardware signer is added, changing the `signer_change_index` ;
- The session owner re-uses the signatures provided earlier, and the sig threshold of 2 is still met ;

Scenario 2:

- Wallet has one hardware signer ;
- Hardware signer and stark signer provide signature for the session ;
- The session is executed and now cached ;
- The wallet removes the strong signer (direct or deferred removal), `signer_change_index` changes ;
- The session owner re-uses the stark signature provided earlier, session can be executed and cached again ;

Recommendation(s): Consider tracking and hashing the specific signer GUIDs for a session, such that if changed, using a subset of the signers will result in a different hash.

Status: Acknowledged

Update from the client: We acknowledge this. We agree that a signature may be reused after cache invalidation but we only see this as a potential issue if account signing strength had increased or changed in such a way that no subset of the existing signature is valid. If an existing signature or its subset is still valid, we think it is ok for a dapp to reuse it without prompting the user.

5.3 [Low] Tokens spent amount cannot be tracked for non-standard methods

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

Description: A session can have spend limits applied to to token addresses, preventing an excessive amount of tokens from being transferred out of the wallet. A token "spend" is identified when a call to a specified token with the following selectors is detected:

```
approve(...)
increase_allowance(...)
increaseAllowance(...)
```

While these functions listed above cover the primary ways to allow ERC20 asset transfers, many popular tokens have non-standard functions which also allow asset transfer. An example of such a token is `zkLend: zETH` which is the second most-held token on Starknet, and has additional methods like `felt_transfer(...)` and `transfer_all(...)`. This makes it possible for some tokens to be transferred out of the wallet even if a spend limit is applied, however it should be noted that the non-standard transfer function would still need to be specifically allowed in the first place.

Recommendation(s): Given that the standard transfer functions are supported by all ERC20 tokens, the need for a session to support one of the non-standard functions is very low. If support for all transfer functions is required, then a redesign of the token spend tracking will be required to handle arbitrary transfer functions. However, if the wallet only needs to support the standard transfer functions, then no code changes is needed, but during session creation it should be brought to the users attention that non-standard transfer functions spend amounts cannot be monitored.

Status: Acknowledged

Update from the client: We acknowledge this issue. We will convey this limitation to our users when interacting with session key dApps via the wallet.

5.4 [Info] Deployment via UDC not possible when caller has non-stark signature

File(s): `src/presets/braavos_base_account.cairo`

Description: The `braavos_base_account` has two flows upon deployment in its constructor. If the deployment is from the Braavos account factory then a two-step initialization approach will be done which avoids using the signature field to store configuration data. If the deployment is not from the Braavos account factory then the configuration data is assumed to be in the transaction signature field.

```
#[constructor]
fn constructor(ref self: ContractState, stark_pub_key: StarkPubKey) {
    let caller = get_caller_address();
    if caller != Consts::BRAAVOS_ACCOUNT_FACTORY_ADDR.try_into().unwrap() {
        // ...
        _assert_valid_deploy_params(
            tx_info, stark_pub_key, signature.slice(2, signature.len() - 2)
        );
        // ...
    } else {
        self.initialization_stark_key.write(stark_pub_key);
    }
}
```

If the wallet which initiated the transaction requires a signature length greater than two, then attempting to slice the signature data to be used for `_assert_valid_deploy_params` will include data related to the initiating wallet's signature, causing validation to fail and preventing deployment. This can only happen when attempting to deploy through the Universal Deployer Contract (UDC) as it requires another wallet to start the transaction.

Recommendation(s): As an alternative to the UDC, the Braavos account factory will be introduced which provides an improved way to deploy a new Braavos account through an existing smart contract. No fix is necessary, this finding only highlights an edge-case incompatibility with the UDC however this will be mitigated by the Braavos account factory, which will be the main way to deploy accounts through via existing smart wallets.

Status: Acknowledged

Update from the client: Acknowledged. We address this via the account factory mechanism.

5.5 [Info] Hash preprocessing for SHA-256 does not follow specification

File(s): `src/utils/hash.cairo`

Description: The `add_sha256_padded_input(...)` function implements the preprocessing step of the input to the `SHA-256` function. This process is specified in the chapter 5.1.1 in Secure Hash Standard (SHS) specification. The preprocessing for `SHA-256` input is described as follows:

– append 1 at the end of the message ;
– append padding to length 448 mod 512 ;
– append 64-bit block that is equal to the message length ;

The comments of the `add_sha256_padding(...)` function express exactly that process:

```
/// Adds padding to the input array for SHA-256. The padding is defined as follows:
/// 1. Append a single bit with value 1 to the end of the array.
/// 2. Append zeros until the length of the array is 448 mod 512.
/// 3. Append the length of the array in bits as a 64-bit number.
/// use last_input_word when the number of bytes in the last input word is less than 4.
```

However, the `add_sha256_padding(...)` function does not follow these steps exactly. The function performs the following steps:

– append 1 at the end of the message ;
– append padding to length 480 mod 512 ;
– append 32-bit block that is equal to the message length ;

This can be seen in the code below:

```
fn add_sha256_padding(ref arr: Array<u32>, last_input_word: u32, last_input_num_bytes: u32) {
    let len = arr.len();
    if last_input_num_bytes == 0 {
        arr.append(0x80000000);
    } else {
        let (q, m, pad) = if last_input_num_bytes == 1 {
            (0x100, 0x1000000, 0x800000)
        } else if last_input_num_bytes == 2 {
            (0x10000, 0x10000, 0x8000)
        } else {
            (0x1000000, 0x100, 0x80)
        };
        let (_, r) = crate::integer::u32_safe_divmod(last_input_word, q);
        arr.append(r * m + pad);
    }
    // assume arr.len() == 1, then remaining is equal to 14, which results in appending 14 zero words
   // in result giving length 15 * 32 = 480 mod 512
    let mut remaining: felt252 = 16 - ((arr.len() + 1) % 16).into();

    append_zeros(ref arr, remaining);
    // the length is added as u32 number, which results in appending the length as 32-bit not 64-bit block
    arr.append(len * 32 + last_input_num_bytes * 8);
}
```

The implementation does not follow the official standard, which is a bad practice. If this is an optimization choice, it should be clearly stated in the comments. Additionally, the maximum length of the input is much lower (32-bit length) than in the standard (64-bit length). Note that this may not be practically important since this is large data to submit on-chain.

Recommendation(s): Consider following the standard in the preprocessing. Alternatively, consider clearly documenting in the comments why the implementation deviates from the standard.

Status: Acknowledged

Update from the client: Acknowledged. Comments in the code were updated to inform of this anomaly.

5.6 [Info] Missing length check during stark signature validation

File(s): src/presets/braavos_base_account.cairo, src/presets/braavos_account.cairo

Description: When processing signatures, there should be a check to ensure that the length of the signature is not excessive. Some signature validations in the Braavos account are done by assuming that the signature is a Stark signature (where the length would be 2) and directly grabbing the first and second elements of a signature span, without checking for the presence of any additional elements.

```
// `sig_1` and `sig_2` are extracted, but signature size can be of arbitrary length
// No validation on the additional fields or checks on span length
[sig_1, sig_2, extra_3, extra_4, extra_5, ...]
```

This poses a potential risk to wallet owners as they may post a transaction providing valid signature data and a malicious actor could extract the valid two elements in the signature, add additional data to inflate the signature size and then frontrun the user's transaction. When the transaction is executed the two valid signature elements will be parsed and the signature will be valid, with the additional data ignored. Since signature data incurs a gas cost based on its size, the malicious actor could significantly increase the gas cost of the transaction.

The Stark signature validation function is used in many areas of the codebase, some of which include length checks before the internal `validate_signature` function is called, however in the functions `braavos_base_account::__validate__` and `sessions::_validate_session_execute` no length check is present, meaning it's possible for a malicious actor to frontrun and inflate gas costs through a factory deployments, or session execution. The validation function is shown below:

```
fn validate_signature(self: @StarkPubKey, hash: felt252, signature: Span<felt252>) -> bool {
    // @audit Length is not checked, additional fields can be added and signature will be valid
    check_ecdsa_signature(hash, *self.pub_key, *signature.at(0), *signature.at(1))
}
```

Recommendation(s): Consider adding a signature length check to the function `StarkSignerMethods::validate_signature`.

Status: Acknowledged

Update from the client: Acknowledged. This will be fixed in next version prior to any fee market implementation in Starknet.

Update from Nethermind Security: This issue has been addressed with a fix for the `braavos_base_account::__validate__` function as it was identified earlier in the audit process, but the `sessions::_validate_session_execute` will be addressed in the next version.

6 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Braavos Wallet documentation

The documentation for the Braavos Account Wallet is contained in the README of the project's Github. It is written for developers, presenting each core component of the wallet, including deployment, signers, multisig, deferred removals, daily withdrawal limits, and outside execution.
The information is presented in a very concise and technical manner, with well-written explanations and references where necessary. It also features a section explaining development dependencies and instructions on how to build and test the code.
Code comments are also of high quality, with explanations for every function describing the purpose, context, and situations where the function will be called. Other than functions, some comments exist in other areas of the code where extra information is necessary, allowing readers to understand the codebase at a faster pace.

7 Test Suite Evaluation

7.1 Compilation Output

```
>$ scarb build -v
    Compiling braavos_account v1.0.1 (/NM-0291-Security-Review-Braavos-Session-Keys/Scarb.toml)
    Finished `dev` profile target(s) in 21 seconds
```

7.2 Tests Output

```
% pytest
============================================= test session starts ==============================================
...
collected 646 items
...
======================================= 646 passed in 2150.96s (0:35:50) =======================================
```

8 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 procehttps://www.overleaf.com/project/65c0e737f41a29601bda5c48ss, 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