Security Review Report
NM-0471 RosettaNet Contracts

NETHERMIND
SECURITY

(September 22, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security on the RosettaNet smart contracts. RosettaNet is a combination of infrastructure and smart contract logic that allows a user to interact with the Starknet network using a regular EVM wallet and a custom RPC url. Transactions are forwarded to the Starknet network where a 'RosettaAccount' can be created on behalf of the user, with EVM based signature verification and built-in features for decoding EVM calldata into `felt252` based Cairo calldata.

The audit comprises 3254 lines of Cairo 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 7 points of attention, where three are classified as `High`, two are classified as `Low` and two are classified as `Info` or `Best Practices` severity. The issues are summarized in Fig. 1.

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

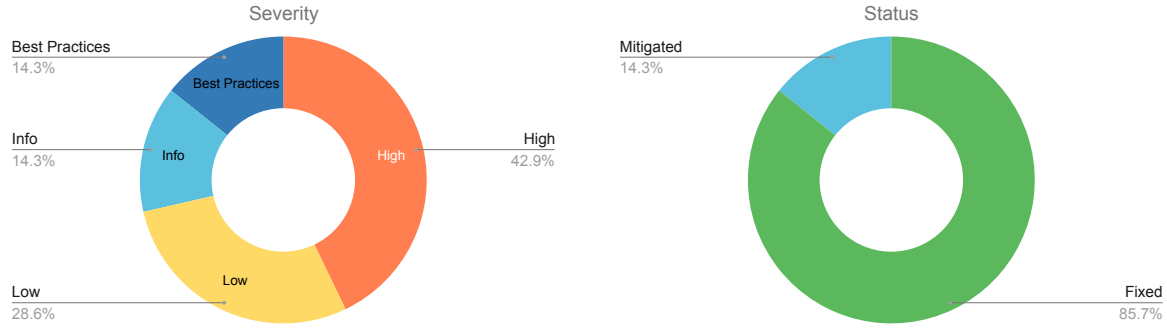

Fig. 1: Distribution of issues: Critical (0), **High** (3), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (1), **Best Practices** (1). **Distribution of status: Fixed** (6), **Acknowledged** (0), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	June 05, 2025
Final Report	September 22, 2025
Repositories	`digine-labs/rosettacontracts`
Initial Commit	`27b7b759cf2391b7db56d67ce6be1a1610a4761c`
Final Commit	`10e93f8a09f05863b42b286a30c535cbeeacb306`
Documentation	`README.md`
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/rosettanet.cairo	254	78	30.7%	55	387
2	src/utils.cairo	79	5	6.3%	21	105
3	src/optimized_rlp.cairo	466	39	8.4%	85	590
4	src/constants.cairo	78	0	0.0%	4	82
5	src/lib.cairo	20	0	0.0%	6	26
6	src/utils/bytes.cairo	225	21	9.3%	39	285
7	src/utils/bits.cairo	147	1	0.7%	12	160
8	src/utils/decoder.cairo	1121	91	8.1%	158	1370
9	src/components/utils.cairo	32	1	3.1%	11	44
10	src/components/function_registry.cairo	67	10	14.9%	18	95
11	src/accounts/utils.cairo	239	8	3.3%	39	286
12	src/accounts/errors.cairo	33	0	0.0%	0	33
13	src/accounts/multicall.cairo	160	3	1.9%	22	185
14	src/accounts/types.cairo	22	3	13.6%	3	28
15	src/accounts/base.cairo	311	19	6.1%	56	386
	Total	**3254**	**279**	**8.6%**	**529**	**4062**

3 Summary of Issues

	Finding	Severity	Update
1	Important transaction fields are not protected by signature	High	Fixed
2	Tuple `has_dynamic` check does not cover all dynamic types	High	Fixed
3	Tuple `has_dynamic` check doesn't handle nested dynamic tuples	High	Fixed
4	Ethereum zero address can point to deployable non-zero Starknet address	Low	Fixed
5	Missing bounds checks for `uint` and `int` decoding	Low	Fixed
6	Double writes in `register_function` can corrupt stored serialized data	Info	Mitigated
7	Inconsistent use of errors in `RosettaAccount`	Best Practices	Fixed

4 System Overview

RosettaNet is a combination of infrastructure and smart contracts that allow regular EVM wallets to interact with Starknet via a custom RPC url. On the Starknet layer RosettaNet consists of **RosettaAccounts** and a **Registry**. The following is a description for each contract:

RosettaAccount: A Starknet account abstraction implementation with custom signature verification to accept Ethereum based ECDSA signatures. These accounts accepts EVM based calldata, and rely on the registry contract to receive information on how to properly decode the data into calldata compatible with Starknet protocols. Multicalls within one transaction are supported by combining multiple EVM calls and structuring the calldata appropriately. Accounts can be upgraded via a "feature call" where it will call the registry contract to get the latest RosettaNet account class hash to upgrade to.

Registry: Acts as a factory for RosettaNet accounts, allowing deployment and registration for accounts deployed outside of the regular flow. Protocols can regitser themselves to RosettaNet, and EVM addresses can be mapped to Starknet addresses allowing for the proper conversion of EVM to Starknet calldata by the accounts. EVM based function signatures are also mapped to Starknet entrypoints, allowing functions and their arguments to be properly decoded into the correct format, all computed onchain.

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] Important transaction fields are not protected by signature

File(s): src/accounts/base.cairo

Description: The data that RosettaNet verifies during signature checks are different to most account contract wallet implementations on Starknet due to the requirement for EVM compatibility. Instead of verifying the signature against the Starknet-layer transaction hash via `get_tx_info`, it is verified against the hash of the EVM transaction within the calldata. This means that some important `V3` transaction fields are not protected by the signature, unless indirectly linked via a check to enforce that the transaction field is constrained to data within the `RosettaNet` call.

The following is a breakdown of all fields in a `V3` transaction, with an explanation of what fields are protected by the signature:

```
fee_data_availability_mode                  <- sn: always zero
nonce                                       <- sn: network protected
nonce_data_availability_mode                <- sn: always zero
paymaster_data                              <- sn: always empty
resource_bounds[L1_GAS].max_amount          <- bound by call.gas_limit
resource_bounds[L1_GAS].max_price_per_unit  <- bound by call.gas_price | max_fee_per_gas
resource_bounds[L1_DATA].max_amount         <- NOT PROTECTED
resource_bounds[L1_DATA].max_price_per_unit <- NOT PROTECTED
resource_bounds[L2_GAS].max_amount          <- NOT PROTECTED
resource_bounds[L2_GAS].max_price_per_unit  <- NOT PROTECTED
signature                                   <- N/A
tip                                         <- sn: always zero
version                                     <- bound to be v3
```

The `chain_id` field is not protected by a signature, meaning this field can be changed and the transaction signature will still be considered valid. An attacker could target users on Sepolia who have an RosettaNet nonce higher than their account on Mainnet, and replay their Sepolia transactions on Mainnet just by changing the `chain_id` field as long as the nonces align with the requirements of the Mainnet account.

The resource bound fields for dict entries `L1_DATA` and `L2_GAS` are also not protected by the signature. The `RosettaAccount` only checks for `L1_GAS` entries and ensures they are bound by the `RosettaCall` gas related fields, as shown below:

```
fn validate_resources(...) {
    let tx_info = get_tx_info().unbox();
    let resource_bounds: Span<ResourcesBounds> = tx_info.resource_bounds;
    for resource in resource_bounds {
        if (*resource.resource == 'L1_GAS') {
            assert(*resource.max_amount == max_amount, MAX_AMOUNT_WRONG);
            assert(
                *resource.max_price_per_unit == max_price_per_unit,
                MAX_PRICE_UNIT_WRONG,
            );
        }
    }
}
```

The `L1_DATA` and `L2_GAS` fields can contribute to overall gas limits for a transaction, and without being protected by the signature it is possible for these fields to be manipulated and increase the gas limits beyond expected amounts. This could potentially lead to a loss of STRK tokens where the gas limit is much higher than expected, allowing more gas to be consumed before reverting the transaction. Furthermore, according to the Starknet docs all fields must be checked and have bounds specified for Starknet v0.14.0.

While unlikely, these separate field issues could potentially be combined by searching for a transaction on Sepolia which fails due to an unbound loop reaching the gas limit. The transaction could then be replayed on mainnet with extremely high modified gas limits to expend a large amount of user STRK tokens.

Recommendation(s): Consider protecting the `chain_id` and all entries in `resource_bounds` to the signature to prevent cross-network transaction replays and gas limit tampering.

Status: Fixed

Update from the client: The `ChainID` will have a unique constant for mainnet and testnet deployments, and gas fields are now protected.

6.2 [High] Tuple `has_dynamic` check does not cover all dynamic types

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

Description: Tuples can be arranged in calldata in two different ways, depending on whether the tuple contains dynamic or fixed-size data. A fixed-size tuple has all elements inlined with other arguments, and does not require an offset or pointer to indicate where the struct is located. A dynamic tuple uses an offset to point the actual location of the data, and the offsets for the dynamic data within the tuple data uses a relative offset from the tuple location data rather than an absolute offset.

```
struct Fixed {
    uint256 one;
    uint256 two;
}
struct Dynamic {
    uint256 one;
    uint256[] two;
}

function processFixed(uint256 a, Fixed calldata b, uint256 c);      // No offset needed
function processDynamic(uint256 a, Dynamic calldata b, uint256 c);  // Requires offset

/*
Calldata for processFixed(1, Fixed(2,3), 4)
0000000000000000000000000000000000000000000000000000000000000001    // arg a
0000000000000000000000000000000000000000000000000000000000000002    // arg b.one
0000000000000000000000000000000000000000000000000000000000000003    // arg b.two
0000000000000000000000000000000000000000000000000000000000000004    // arg c


Calldata for processDynamic(1, Dynamic(2, [3,4]), 5)
0000000000000000000000000000000000000000000000000000000000000001    // arg a
0000000000000000000000000000000000000000000000000000000000000060    // arg b offset
0000000000000000000000000000000000000000000000000000000000000005    // arg c
0000000000000000000000000000000000000000000000000000000000000002    // arg b.one
0000000000000000000000000000000000000000000000000000000000000040    // arg b.two relative offset
0000000000000000000000000000000000000000000000000000000000000002    // arg b.two length
0000000000000000000000000000000000000000000000000000000000000001    // arg b.two[0]
0000000000000000000000000000000000000000000000000000000000000002    // arg b.two[1]

*/
```

When decoding a tuple in `decode_tuple` there is a check to identify whether a tuple is dynamic or fixed-size, as a unique approach is required for each. This check is done using `has_dynamic` which checks for dynamic data types within the tuple:

```
fn has_dynamic(types: Span<EVMTypes>) -> bool {
    let mut result = false;
    for evm_type in types {
        match evm_type {
            EVMTypes::Array => { result = true; },
            // Missing EVMTypes::Bytes
            // Missing EVMTypes::String
            _ => { continue; },
        };
    };

    result
}
```

The function only considers the type `EVMTypes::Array` to be dynamic, however, the following types `EVMTypes::Bytes` and `EVMTypes::String` are also dynamic but are not included as part of the check. This can lead to incorrect calldata decoding where `decode_tuple` may incorrectly assume a tuple is fixed-size and parse the offset value and calldata belonging to other arguments as struct data. As a result, execution may revert or corrupted data may be returned from the function.

Recommendation(s): Consider adding `EVMTypes::Bytes` and `EVMTypes::String` to the `has_dynamic` check to ensure that all dynamic tuples can be decoded correctly.

Status: Fixed

Update from the client: Tuples and strings are now considerd dynamic.

6.3 [High] Tuple `has_dynamic` check doesn't handle nested dynamic tuples

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

Description: As described in the finding above, tuples can be dynamic or fixed-size, depending on the data they contain. If you have an inner tuple which has dynamic data fields, and an outer tuple which contains the inner tuple, the outer tuple is also dynamic. The `has_dynamic` check only checks the tuple fields one layer deep, meaning that the outer tuple describe above would be considered as fixed-size even though it contains a dynamic tuple. The function is shown below:

```
fn has_dynamic(types: Span<EVMTypes>) -> bool {
    let mut result = false;
    for evm_type in types {
        match evm_type {
            EVMTypes::Array => { result = true; },
            // An inner tuple may be dynamic, but this is not checked
            _ => { continue; },
        };
    };

    result
}
```

When a tuple is treated as fixed-size when it is actually dynamic, the decoding logic will parse the offset values and other argument values as part of the struct data. As a result, the function `decode_tuple` will either revert, or corrupted data may be returned.

Recommendation(s): Consider recursively checking all inner tuples in `has_dynamic` to ensure that nested dynamic tuple decoding behaves correctly.

Status: Fixed

Update from the client: The `has_dynamic` check now can handle recursive tuples.

6.4 [Low] Ethereum zero address can point to deployable non-zero Starknet address

File(s): src/rosettanet.cairo

Description: If a user intends to interact with the Ethereum zero address using RosettaNet, there are two possible functions used to determine which associated Starknet address should be used: `get_starknet_address` or `get_starknet_address_with_fallback`. The first directly querying the mapping `sn_to_eth` and returning the entry, and the second introducing a fallback feature that can calculate the expected associated Starknet address even if there is no association in the mapping. The function is shown below:

```
fn get_starknet_address_with_fallback(
    self: @ContractState, eth_address: EthAddress,
) -> ContractAddress {
    let address_on_registry: ContractAddress = self.eth_to_sn.entry(eth_address).read();
    if (address_on_registry.is_zero()) {
        return self.precalculate_starknet_account(eth_address);
    }
    address_on_registry
}
```

In a scenario where a user has the Ethereum zero address in their calldata, the RosettaNet decoder will query the registry contract using `get_starknet_address_with_fallback` and since there would be no association, it will call `precalculate_starknet_account`, which will result in a nonzero Starknet address. Protocols which integrate RosettaNet and have special meanings for a zero address may not behave correctly.

Consider a protocol which interprets a transfer to the zero address as a burn, and is programmed to reduce the total supply. The EVM calldata will contain a zero address and after RosettaNet decoding, the new recipient will be a nonzero Starknet address. As a result, the total supply will remain unchanged even though the expected behavior would have been for the total supply to decrease.

Additionally, it is possible to deploy the RosettaNet account associated with the Ethereum zero address, as there are no checks in the `RosettaAccount::constructor` to prevent the `ethereum_address` argument from being zero. This means that even the non-fallback `get_starknet_address` function can return a non-zero Starknet address when called with an Ethereum zero address input.

```
#[constructor]
fn constructor(ref self: ContractState, eth_address: EthAddress, registry: ContractAddress) {
    // `eth_address` argument can be zero
    self.ethereum_address.write(eth_address);
    self.registry.write(registry);
}
```

Recommendation(s): Consider adding additional logic to `get_starknet_address_with_fallback` to immediately return the Starknet zero address if an Ethereum zero address is provided. Also consider adding a check to `RosettaAccount::constructor` to prevent the `eth_address` argument from being zero.

Status: Fixed

Update from the client: All address getter functions now have a zero address check, and zero addresses cannot be registered.

6.5 [Low] Missing bounds checks for `uint` and `int` decoding

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

Description: The decode functions `decode_uint` and `decode_int` are missing checks to ensure that the data is within the expected size. While this has no impact on the RosettaNet protocol or decoding itself, it may have an impact on protocols which are on the registry, as they may not feature such bounds checks for non-256-bit values which may cause their logic to behave incorrectly.

```
#[inline(always)]
fn decode_uint(ref ctx: EVMCalldata, size: u32) -> Span<felt252> {
    // Missing range check
    let (new_offset, value) = ctx.calldata.read_u256(ctx.offset);
    ctx.offset = new_offset;
    array![value.try_into().unwrap()].span()
}

#[inline(always)]
fn decode_int(ref ctx: EVMCalldata, size: u32) -> Span<felt252> {
    // Missing range check
    let (new_offset, value) = ctx.calldata.read_u256(ctx.offset);
    ctx.offset = new_offset;
    // ...
}
```

This has already been pointed out by the developers in the comments, however it should be addressed before deployment.

Recommendation(s): Consider adding bounds checks to `decode_uint` and `decode_int`.

Status: Fixed

Update from the client: Bounds checks have been added for `uint` and `int` types.

6.6 [Info] Double writes in `register_function` can corrupt stored serialized data

File(s): src/components/function_registry.cairo

Description: The function `register_function` is used to associated an ETH function selector with a series of inputs which is used for decoding. There are no checks to ensure that an existing selector has not already been written before. In this case, the data will be blindly appended ontop of the existing data, corrupting the serialized EVM decoding information. This will cause `EVMType` decodes to revert on the account contracts which query the registry. If this occurs it is not possible to modify these data entries without a class hash upgrade.

```
fn register_function(
    ref self: ComponentState<TContractState>, fn_name: ByteArray, inputs: Span<EVMTypes>,
) {
    // ...
    // No checks to see if this vector already contains data
    let vector_serialized_directives = self.directives.entry(eth_selector);
    for i in 0..inputs_serialized.len() {
        // New data is blindly appended
        vector_serialized_directives.append().write(*inputs_serialized.at(i));
    };
    // ...
}
```

Recommendation(s): Consider adding a check to ensure the selector doesn't already contain serialiazed decode data.

Status: Mitigated

Update from the client: Access control has been added so only privileged addresses can register, but double writes are still possible.

6.7 [Best Practices] Inconsistent use of errors in `RosettaAccount`

File(s): `src/accounts/base.cairo`

Description: The majority of `assert` and `expect` error strings in the `RosettaAccount` contract are specified in `AccountErrors` as constants. However, there are some error messages within `__validate_deploy__` which break this pattern by having the strings written directly:

```
fn __validate_deploy__(...) -> felt252 {
    assert(..., 'Salt and param mismatch');
    assert(..., 'registry zero');
    // ...
    assert(..., 'deployed address wrong');
    // ...
}
```

Recommendation(s): Consider defining these errors as constants in `AccountErrors` to remain consistent with other `RosettaAccount` errors.

Status: Fixed

Update from the client: Error codes are now used for the mentioned `assert` statements.

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 CandideLabs documentation

The documentation provided for the RosettaNet contracts are concise, with a `README.md` document explaining the protocol on a high level. During the audit engagement a kick-off call was held with an explanation from the RosettaNet team on how the protocol operates. In addition to sync calls, there has been regular asynchronous communication via Telegram to explain new issues and clarify questions or concerns.

8 Test Suite Evaluation

8.1 Compilation Output

```
> scarb --version
scarb 2.10.1 (ea6b66386 2025-02-20)
cairo: 2.10.1 (https://crates.io/crates/cairo-lang-compiler/2.10.1)
sierra: 1.7.0

> snforge --version
snforge 0.38.3

> scarb build
   Compiling snforge_scarb_plugin v0.38.3
    Finished `release` profile [optimized] target(s) in 0.03s
   Compiling rosettacontracts v0.1.0 (/Users/kalzak/dev/nm/gauss/audits/0471/rosettacontracts/Scarb.toml)
    Finished `dev` profile target(s) in 12 seconds
```

8.2 Tests Output

```
> snforge test

Collected 105 test(s) from rosettacontracts package
Running 60 test(s) from src/
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_list_short_multi
[PASS] rosettacontracts::optimized_rlp::tests::test_byte_size
[PASS] rosettacontracts::optimized_rlp::tests::test_keccak_example
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int128_neg
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes_one_slot
[PASS] rosettacontracts::optimized_rlp::tests::step_cost_opt_rlp_long_string
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int8_neg
[PASS] rosettacontracts::utils::decoder::tests::test_decode_complex
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_list_long_multi
[PASS] rosettacontracts::accounts::multicall::tests::test_prepare_multicall_context
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int8_pos
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int256_neg
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int256_pos
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_list_short
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int256_zero
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_bytearray_empty
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_uint128s
[PASS] rosettacontracts::utils::decoder::tests::test_decode_int8_zero
[PASS] rosettacontracts::utils::decoder::tests::test_decode_multi_uint128
[PASS] rosettacontracts::utils::decoder::tests::test_decode_multi_uint256
[PASS] rosettacontracts::utils::decoder::tests::test_decode_tuple_array
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_bytearray_long
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes2
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_bytearray_low
[PASS] rosettacontracts::utils::decoder::tests::test_decode_tuple_array_static_first
[PASS] rosettacontracts::utils::decoder::tests::test_decode_tuple_of_two
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes2_zero
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_of_struct
[PASS] rosettacontracts::utils::decoder::tests::test_decode_tuple_of_two_uint256
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_short_string
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_tuple_with_multi_arrays
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes31
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_eip1559_tx_no_calldata
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes32_zeroes
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_bytearray_mid
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_bytearray_very_long
[PASS] rosettacontracts::utils::decoder::tests::test_decode_felt252
[PASS] rosettacontracts::utils::decoder::tests::test_decode_felt252_max
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_legacy_tx_calldata_long
[PASS] rosettacontracts::accounts::utils::tests::test_calldata_conversion_long
[PASS] rosettacontracts::accounts::utils::tests::test_rlp_encode_legacy
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes32
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_list_empty
```

```
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_list_long
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_short_string_lower
[PASS] rosettacontracts::optimized_rlp::tests::step_cost_opt_rlp
[PASS] rosettacontracts::accounts::utils::tests::test_generate_legacy_tx_hash
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_short_empty
[PASS] rosettacontracts::accounts::utils::tests::test_generate_eip1559_tx_hash
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_of_strings_long_elements
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_tuple_statics_dynamic_on_middle
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_of_array
[PASS] rosettacontracts::utils::decoder::tests::test_decode_felt252_max_higher
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_example_calldata
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_encode_legacy_tx_no_calldata
[PASS] rosettacontracts::optimized_rlp::tests::test_rlp_short_string_zero
[PASS] rosettacontracts::utils::decoder::tests::test_decode_bytes_one_full_slot
[PASS] rosettacontracts::utils::decoder::tests::test_decode_array_of_strings
[PASS] rosettacontracts::accounts::utils::tests::test_calldata_conversion
Running 45 test(s) from tests/
[IGNORE] rosettacontracts_tests::account_tests::test_execute_multicall_transaction
[IGNORE] rosettacontracts_tests::rosettanet_tests::rosettanet_redeploy_same_account
[PASS] rosettacontracts_tests::account_tests::test_legacy_multicall_validation
[PASS] rosettacontracts_tests::account_tests::test_execute_erc20_transfer_exceeds_balance
[PASS] rosettacontracts_tests::account_tests::test_execute_value_transfer_and_call
[PASS] rosettacontracts_tests::account_tests::test_legacy_transaction_validation_value_transfer_only
[PASS] rosettacontracts_tests::account_tests::test_eip1559_transaction_validation_calldata
[PASS] rosettacontracts_tests::account_tests::test_execute_value_transfer_not_enough_balance
[PASS] rosettacontracts_tests::account_tests::test_execute_erc20_transfer
[PASS] rosettacontracts_tests::account_tests::test_execute_erc20_transfer_receiver_not_registered
[PASS] rosettacontracts_tests::account_tests::test_transaction_validation_unsupported_tx_type
[PASS] rosettacontracts_tests::account_tests::test_eip1559_transaction_validation_first_transaction_different_nonce
[PASS] rosettacontracts_tests::account_tests::test_eip1559_transaction_validation_value_transfer_only
[PASS] rosettacontracts_tests::account_tests::test_eip1559_validate_second_nonce
[PASS] rosettacontracts_tests::account_tests::test_legacy_transaction_wrong_gas
[PASS] rosettacontracts_tests::account_tests::test_legacy_transaction_validation_calldata_invalid_signature
[PASS] rosettacontracts_tests::account_tests::test_multicall_wrong_selector
[PASS] rosettacontracts_tests::account_tests::test_multicall_with_value
[PASS] rosettacontracts_tests::function_registry_tests::test_register_function
[PASS] rosettacontracts_tests::account_tests::test_execute_erc20_transfer_legacy
[PASS] rosettacontracts_tests::function_registry_tests::test_register_function_initial_byte_zero
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_deploy_initial_dev
[PASS] rosettacontracts_tests::account_tests::test_signature_validation_legacy
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_register_non_deployed_contract
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_check_precalculated_address
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_non_dev_set_class
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_register_contract
[PASS] rosettacontracts_tests::account_tests::test_transaction_validation_calldata_and_value_transfer
[PASS] rosettacontracts_tests::account_tests::test_multicall_validate_actual_values
[PASS] rosettacontracts_tests::account_tests::test_signature_validation_legacy_invalid
[PASS] rosettacontracts_tests::account_tests::test_validate_multicall_transaction
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_register_existing_contract
[PASS] rosettacontracts_tests::account_tests::test_legacy_transaction_wrong_tx_version
[PASS] rosettacontracts_tests::account_tests::test_legacy_transaction_validation_calldata
[PASS] rosettacontracts_tests::account_tests::test_execute_value_transfer_wrong_value_on_sig
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_set_class
[PASS] rosettacontracts_tests::test_utils::test_storage_manipulation
[PASS] rosettacontracts_tests::test_utils::test_deploy_rosettanet
[PASS] rosettacontracts_tests::account_tests::test_validate_multicall_transaction_wrong_signature
[PASS] rosettacontracts_tests::account_tests::test_validation_real_data_failing
[PASS] rosettacontracts_tests::rosettanet_tests::rosettanet_register_account_class_as_contract
[PASS] rosettacontracts_tests::account_tests::test_execute_value_transfer
[PASS] rosettacontracts_tests::account_tests::test_signature_validation_eip1559
[PASS] rosettacontracts_tests::account_tests::test_signature_validation_eip1559_invalid
[PASS] rosettacontracts_tests::account_tests::test_legacy_transaction_wrong_nonce
Tests: 103 passed, 0 failed, 0 skipped, 2 ignored, 0 filtered out
```

9 About Nethermind

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

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

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

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

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

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

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

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

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

Learn more about us at nethermind.io.

General Advisory to Clients

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

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.