Security Review - Report
NM-0062: EveryRealm

NETHERMIND

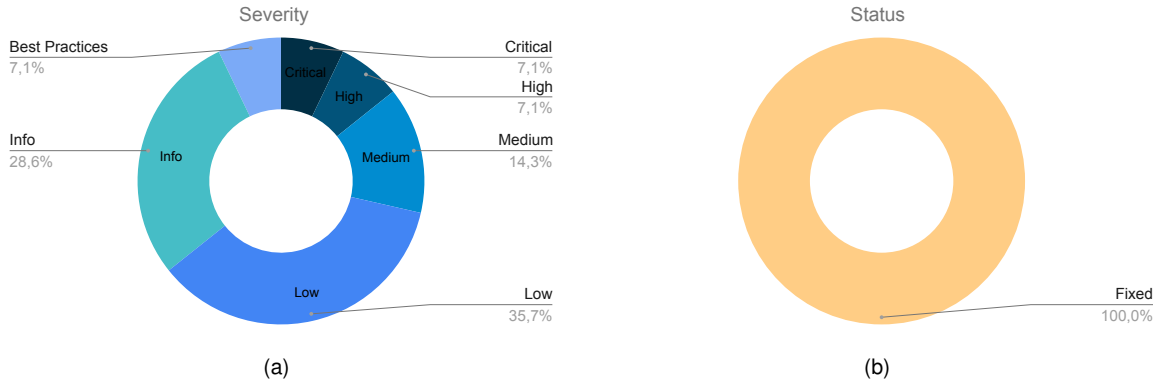
(Oct 13, 2022)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the EveryRealm Signed Executor Tokens written in the Solidity language. The Signed Executor Tokens are ERC-721 and ERC-1155 tokens that support owners authorizing certain actions through signatures, allowing the recipient or any other third party to execute the transaction with the signature provided by the owner. The message signing and verification implementation makes use of the EIP-712 standard to sign token owner actions, which can be verified and executed by any user that provides a valid signature. **This code is the result of the investigation by the EveryRealm team into the idea of allowing signatures to be used for executing token actions**. Currently the only actions supported by signature execution are token transfers and approvals, however support for more token actions is planned in the future.

During the initial audit, we have raised 14 points of attention. **During the reaudit**, the EveryRealm team has agreed with the reported issues and they are working on the fixes. **The codebase is composed of** 200 lines of Solidity code with a test suite composed of 14 tests. **The distribution of issues** is summarized in the figure below. **This document is organized as follows:** Section 2 presents files in the scope for this audit. Section 3 summarizes the findings in a table. Section 4 discusses the risk rating methodology adopted for this audit. Section 5 details each finding. Section 6 discusses the documentation provided for this audit. Section 7 discusses the automated test suite and presents test output. Section 8 concludes the audit report.



(a) (b)

Distribution of Issues: Critical (1), **High** (1), **Medium** (2), **Low** (5), **Informational** (4), **Best Practices** (1).
Distribution of Status: Fixed (14), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (14)

Summary of the Audit

Audit Type	Security Review
Initial Report	Oct. 13, 2022
Response from Client	Oct. 18, 2022
Final Report	Oct. 25, 2022
Methods	Manual Review, Automated Analysis
Repository	EveryRealm
Commit Hash (Initial Audit)	44f3b9f76b5300dc371cc2d547a7a9b4bf3313b0
Commit Hash (Final Audit)	Fixes under implementation
Documentation	Not provided
Unit Tests Assessment	Medium

2 Contracts

	Contract	Lines of Code	Lines of Comments	Comments Ratio	Blank Lines	Total Lines
1	contracts/GenerativeNFT.sol	67	3	4.5%	16	86
2	contracts/Mock1155.sol	7	0	0.0%	2	9
3	contracts/SignedExecutor721.sol	60	5	8.3%	16	81
4	contracts/Mock721.sol	7	0	0.0%	2	9
5	contracts/SignedExecutor1155.sol	59	1	1.7%	10	70
	Total	**200**	**9**	**4.5%**	**46**	**255**

3 Summary of Issues

	Finding	Severity	Update
1	Not guarding against `Signature Replay Attacks`	Critical	Fixed
2	Not guarding against `Signature Replay Attacks`	High	Fixed
3	The `mintPerTx` limit can be bypassed	Medium	Fixed
4	Total collection size cannot reach maximum	Medium	Fixed
5	Missing input validation for `mintPerTx`	Low	Fixed
6	Missing input validation for `totalSupplyLimit`	Low	Fixed
7	Missing input validation in `mint(...)`	Low	Fixed
8	Sale parameters should be immutable while sale is active	Low	Fixed
9	`transfer(...)` can fail to send Ether	Low	Fixed
10	Events emitted using storage variable data	Info	Fixed
11	Missing re-entrancy guard in `mint(...)`	Info	Fixed
12	Multiple Solidity versions	Info	Fixed
13	Redundant zero initialization on storage variables	Info	Fixed
14	`changeIsSaleActive` might not work as intended	Best Practices	Fixed

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 is a measure of 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 the finding were to be exploited by an attacker. 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, 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 General

5.1.1 [Info] Multiple Solidity versions

File(s): `contracts/*`

Description:

Both of the `SignedExecutor` contracts use the Solidity version `0.8.17` while the `GenerativeNFT` contract uses the Solidity version `0.8.16`. Using the same Solidity versions throughout a codebase is recommended to improve consistency and reduce the risk of compatibility issues. All contracts also use a floating pragma. Contracts should be deployed using the same compiler version which they have been tested with. Locking the pragma ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs.

Recommendation(s): Consider removing the floating pragma and using the same Solidity version for all contracts.

Status: Fixed

Update from the client: Agree. Versions updated to use 0.8.17 across all contracts.

5.2 `contracts/GenerativeNFT.sol`

5.2.1 [Medium] The `mintPerTx` limit can be bypassed

File(s): `contracts/GenerativeNFT.sol`

Description: The function `mint(...)` ensures that the number of tokens minted in a single transaction cannot exceed a set limit. The user passes an argument `quantity` to indicate how many tokens should be minted. If `quantity` is greater than `mintPerTx` then the call will revert. This approach does not count the amount of tokens minted in the current transaction, it instead counts the amount minted in the current call. An attacker could deploy a contract which makes multiple separate calls to `mint(...)` where each call would pass a check, even though the total amount of tokens minted in the transaction is higher than the imposed limit.

Recommendation(s): Consider changing the mint limit implementation to track the number of tokens minted across the entire transaction rather than per unique call.

Status: Fixed

Update from the client: Agree. `mintPertx` to be removed.

5.2.2 [Medium] Total collection size cannot reach maximum

File(s): `contracts/GenerativeNFT`

Description: In the function `mint(...)`, the total supply check uses the less-than (<) operator to ensure that the total supply will never reach `totalSupplyLimit`.

In the function `mint(...)` a check is done to ensure that the amount to be minted will not cause the total supply to exceed the total supply limit. This check is done with the less-than (<) operator, which means that `totalSupply` will never reach `totalSupplyLimit`. The total supply check is shown below:

```
function mint(uint256 quantity) external payable {
    require(quantity + totalSupply < totalSupplyLimit, "mint: Exceeding total collection size.");
    ...
}
```

For example, with collection size `totalSupplyLimit = 100` and current `totalSupply = 99`, calling `mint(quantity = 1)` will be reverted.

Recommendation(s): Consider changing < to <= in the total supply check.

Status: Fixed

Update from the client: Agree. Check if `totalSupply` is greater than or equal to `totalSupply` to be implemented.

5.2.3 [Low] Missing input validation for `mintPerTx`

File(s): `contracts/GenerativeNFT`

Description: The storage variable `mintPerTx` can be set in the constructor and in the function `changeMintPerTx(...)`. When setting `mintPerTx` there should be a check to ensure the new value cannot be zero.

Recommendation(s): Consider adding input validation to the function `changeMintPerTx(...)` to prevent a new value that is zero from being set. To reduce repeated code, the constructor can call `changeMintPerTx(...)` to use its input validation.

Status: Fixed

Update from the client: Agree. `mintPerTx` functionality to be removed due to lack of utility.

5.2.4 [Low] Missing input validation for `totalSupplyLimit`

File(s): `contracts/GenerativeNFT`

Description: The storage variable `totalSupplyLimit` can be set in the constructor and in the function `changeTotalSupplyLimit(...)`. When setting `totalSupplyLimit` there should be two checks: a) The new value cannot be zero; b) The new value cannot be less than the current total supply.

Recommendation(s): Consider adding input validation to the function `changeTotalSupplyLimit(...)` to prevent a new value that is zero or less than the current total supply. To reduce repeated code, the constructor can call `changeTotalSupplyLimit(...)` to use its input validation.

Status: Fixed

Update from the client: Agree. Validate `totalSupplyLimit` to be greater than current supply to be implemented.

5.2.5 [Low] Missing input validation in `mint(...)`

File(s): `contracts/GenerativeNFT`

Description: There is no check in the function `mint(...)` if the argument `quantity` is higher than 0. When calling this function with `quantity` of zero, no tokens will be minted but an event will still be emitted which can affect protocol logging.

Recommendation(s): Consider adding a check to ensure that `quantity` is not 0.

Status: Fixed

Update from the client: Agree. Validating number to mint to be greater than zero to be implemented.

5.2.6 [Low] Sale parameters should be immutable while sale is active

File(s): `contracts/GenerativeNFT.sol`

Description: The functions `changeTotalSupplyLimit(...)`, `changeMintPerTx(...)` and `changePrice(...)` are callable by the contract owner while a sale is active. To promote trust from users it should not be possible for sale parameters to be modified during a sale, as while a sale is active it should be guaranteed that all users who mint will be under the same conditions as everybody else.

Recommendation(s): Consider implementing a feature to prevent sale parameters from being changed while a sale is live. It should be noted that there should be some restriction on the ability to enable and disable the sale as a simple check to see if a sale is active upon a parameter change can be bypassed by disabling the sale, changing the parameter, then enabling the sale again.

Status: Fixed

Update from the client: Agree. Price and changes to be disabled while sale is active, total supply to be greater than current supply, and `mintPerTx` to be removed.

5.2.7 [Low] `transfer(...)` can fail to send Ether

File(s): `contracts/GenerativeNFT`

Description: In the function `withdraw(...)`, `transfer(...)` is used to send the contract balance. However, `transfer(...)` forwards a fixed amount of gas which is 2,300. This may not be enough to send Ether if the recipient is a contract or gas costs change. Further details can be found here

Recommendation(s): Consider using `call(...)` instead of `transfer(...)` when sending Ether.

Status: Fixed

Update from the client: Agree. Call to be used in place - and check return value is successful.

5.2.8 [Info] Events emitted using storage variable data

File(s): contracts/GenerativeNFT.sol

Description: Some event emissions in the `GenerativeNFT` contract use a storage variable as an argument. This will use an extra `SLOAD` opcode costing `100` gas as it does a "warm access" to an already-written storage variable. In the "change" functions (such as `changePrice(...)`) the calldata argument can be used instead of the storage variable when emitting the event to save this `100` gas.

Recommendation(s): Consider emitting an event using calldata rather than the storage variable to save gas. An example is shown below:

```
function changePrice(uint256 _price) external onlyOwner {
    price = _price;
    emit PriceUpdated(_price);
}
```

Status: Fixed

Update from the client: Agree. Use `calldata` in lieu of memory to be implemented.

5.2.9 [Info] Missing re-entrancy guard in `mint(...)`

File(s): contracts/GenerativeNFT

Description: The function `mint(...)` uses `_safeMint(...)` to mint tokens to the caller address. If the recipient of the mint is a contract address, `_safeMint(...)` will do a callback to the recipient to confirm that they are able to accept the to-be-minted tokens. This opens the opportunity for re-entrancy back into the mint function. Currently this does not pose a risk as the `totalSupply` is updated after all minting is complete, so there will be a `tokenId` collision when minting which will revert. Since this code is currently in development and the function may change, it could be possible in the future for the `totalSupply` to be updated before which would allow for re-entrancy.

Recommendation(s): Consider adding re-entrancy guards to the function `mint(...)` to prevent malicious use of the callback from `_safeMint(...)`.

Status: Fixed

Update from the client: Agree. Reentrant protection to be implemented.

5.2.10 [Info] Redundant zero initialization on storage variables

File(s): contracts/GenerativeNFT

Description: The storage variables `totalSupply`, `isSaleActive` and `baseURI` are already set to default values upon contract initialization. These variables are then set to their default values again, costing more gas than necessary.

Recommendation(s): Consider removing the unnecessary initialization for the variables `totalSupply`, `isSaleActive` and `baseURI`.

Status: Fixed

Update from the client: Agree. Initialization of null values to be removed.

5.2.11 [Best Practices] `changeIsSaleActive` might not work as intended

File(s): contracts/GenerativeNFT.sol

Description: The function `changeIsSaleActive(...)` does not check if the argument `_isSaleActive` is different to the `isSaleActive` storage variable. This can cause an unnecessary event emission and affect protocol logging.

Recommendation(s): Consider creating two functions for setting the sale state: `startSale(...)` and `endSale(...)`. This will make setting sale states a much more straightforward process and prevent unnecessary event emission.

Status: Fixed

Update from the client: Agree. Validate sale state is not the current state to be implemented.

5.3 contracts/SignedExecutor721.sol

5.3.1 [High] Not guarding against Signature Replay Attacks

File(s): contracts/SignedExecutor721.sol

Description: The signatures used in `executeSetIfSignatureMatch(...)` do not make use of a nonce. This opens up the possibility for signature replay attacks as long as the attack is done within the timeframe of the specified deadline.

From the EIP-712 Standard: "This standard is only about signing messages and verifying signatures. In many practical applications, signed messages are used to authorize an action, for example an exchange of tokens. It is very important that implementers make sure the application behaves correctly when it sees the same signed message twice. For example, the repeated message should be rejected or the authorized action should be idempotent. How this is implemented is specific to the application and out of scope for this standard."

Potential Attack: A signature replay attack on an ERC-721 token transfer can allow the same token to be transferred again. If some token was transferred from a user is sent back to their own address, as long as the current time is within the specified deadline this transfer can be successfully executed again without the users permission. An example scenario is described below:

1. Alice wants to sell Bob an NFT for the price of 5 Ether;
2. Alice signs a message to transfer an NFT from her wallet to Bob's wallet with a 24 hour deadline;
3. A trusted third party acts as a middleman and executes the trade;
4. After the trade Bob decides to put the NFT on sale in some marketplace;
5. Alice decides to purchase that NFT back (this purchase is done within the 24 hour deadline);
6. Bob can now use Alice's previous signature to transfer the NFT from Alice again for free;

Recommendation(s): Consider adopting a mechanism to prevent signature replay attacks.

Status: Fixed

Update from the client: Agree and highest priority fix to be implemented. Implementation will use a mapping of address to array of transaction hashes, checking if hash has been seen before executing.

5.4 contracts/SignedExecutor1155.sol

5.4.1 [Critical] Not guarding against Signature Replay Attacks

File(s): contracts/SignedExecutor1155.sol

Description: The signatures used in `executeSetIfSignatureMatch(...)` do not make use of a nonce. This opens up the possibility for signature replay attacks as long as the attack is done within the timeframe of the specified deadline.

From the EIP-712 Standard: "This standard is only about signing messages and verifying signatures. In many practical applications, signed messages are used to authorize an action, for example an exchange of tokens. It is very important that implementers make sure the application behaves correctly when it sees the same signed message twice. For example, the repeated message should be rejected or the authorized action should be idempotent. How this is implemented is specific to the application and out of scope for this standard."

Potential Attack: A signature replay attack on an ERC-1155 is more severe than on an ERC-721 token as explained earlier. This is because an ERC-1155 can act as an ERC-20 token. If a user has 1000 tokens but creates a signature to transfer 100 tokens, this message can be repeated 10 times to transfer the entire amount to the recipient. This means that any token transfer done through `executeSetIfSignatureMatch(...)` can have its signature replayed to continuously transfer tokens from the sender until the underlying `transfer(...)` function fails due to a lack of funds.

Recommendation(s): Consider adopting a mechanism to prevent signature replay attacks.

Status: Fixed

Update from the client: Agree and highest priority fix to be implemented. Implementation will use a mapping of address to array of transaction hashes, checking if hash has been seen before executing.

6 Documentation Evaluation

Technical documentation is designed to explain the software product in a way that developers, users and the whole community can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can appear not only in the form of a `README.md`, but also using **code as documentation**, diagrams, websites, research papers, videos and other sources as well. Specifically, code as documentation is a software engineering practice where the code is written in such a way that it serves as its own explanatory documentation. It can be achieved by using meaningful names for variables, functions and methods following an intuitive design that can be easily understood by readers. Since this code represents an investigation into token signature authorization and is still under development, formal documentation has not yet been made. However, the **code as documentation** principle applies here as the code is clear to read and understand. The code would benefit from NatSpec comments explaining each functions purpose, inputs and outputs. The repository has a `README.md` containing default text generated by Hardhat. Even with experimental code, a high quality `README.md` is recommended to clearly describe the project goals, how to set up the environment, how to compile, how to run tests and any other important notes.

7 Test Suite Evaluation

In this section, we present our evaluation of the test suite provided with the application. We check compilation, tests, and code coverage. The test suite covers expected behavior but does not account for any edge cases. It is important to improve testing to cover for edge cases to identify unexpected behaviors that otherwise would have remained when deployed. In particular, given the nature of this project tests related to signature replay attacks should be considered to identify potential issues. We recommend improving the test suite to achieve 100% code coverage and to expand the testing to cover more edge cases with a focus on message signing and verification.

7.1 Layer 1 Contract Compilation

Below, we present the output of the compilation process. We notice some warnings that must be properly handled.

```
> npx hardhat compile
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/Mock1155.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/Mock721.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.17;"
--> contracts/Mock1155.sol


Warning: Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.17;"
--> contracts/Mock721.sol


Compiled 23 Solidity files successfully
```

7.2 Layer 1 Tests Output

```
> npx hardhat test
  Generative NFT Minting Tests
     Should update total supply limit
     Should update mint per tx
     Should update price
     Should update baseURI
     Should activate Sale
     Should mint 3 NFTs by owner (for free)
     Should mint 3 NFTs
     Should mint 3 NFTs with wrong amount
     Should mint 4 NFTs (not allowed)
     Should try to mint more NFTs than supply limit

  ERC 721 Tests
Account 0 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Account 1 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
     Transfer Signature
     Approval Signature

  ERC 1155 Tests
Account 0 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Account 1 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
     Transfer Signature
     Approval Signature

  14 passing (4s)
```

7.3 Layer 1 Code Coverage

The audited code has not been instrumented with the `Solidity Coverage` tool. In order to use the Solidity Coverage, we must install it using the command line below:

```
npm install --save-dev solidity-coverage
```

After this, we must require the plugin in the `hardhat.config.js` file, as shown below.

```
require('solidity-coverage');
```

After doing these steps, we can check the coverage by running the command:

```
npx hardhat coverage
```

The output is presented below.

```
-------------------------|----------|----------|----------|----------|----------------|
File                     |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-------------------------|----------|----------|----------|----------|----------------|
 contracts/              |    89.58 |    55.77 |    88.24 |    91.67 |                |
  GenerativeNFT.sol      |    81.25 |    53.85 |    77.78 |       88 |       71,75,77 |
  Mock1155.sol           |      100 |      100 |      100 |      100 |                |
  Mock721.sol            |      100 |      100 |      100 |      100 |                |
  SignedExecutor1155.sol |    93.75 |    57.14 |      100 |    94.44 |             68 |
  SignedExecutor721.sol  |    92.86 |    58.33 |      100 |    93.33 |             79 |
-------------------------|----------|----------|----------|----------|----------------|
All files                |    89.58 |    55.77 |    88.24 |    91.67 |                |
-------------------------|----------|----------|----------|----------|----------------|
```

8 About Nethermind

Founded in 2017 by a small team of world-class technologists, Nethermind builds Ethereum solutions for developers and enterprises. Boosted by a grant from the Ethereum Foundation in August 2018, our team has worked tirelessly to deliver the fastest Ethereum client in the market. Our flagship Ethereum client is all about performance and flexibility. Built on .NET core, a widespread, enterprise-friendly platform, Nethermind makes integration with existing infrastructures simple, without losing sight of stability, reliability, data integrity, and security. **Nethermind is made up of several engineering teams across various disciplines, all collaborating to realize the Ethereum roadmap, by conducting research and building high-quality tools**. Teams focus on specific areas of the Ethereum problem space. Each consists of specialists and experienced developers working alongside interns, learning the ropes in the Nethermind Internship Program. **Our mission is to gather passionate talent from around the world, and to tackle some of the blockchain's most complex problems**. Nethermind provides software solutions and services for developers and enterprises building the Ethereum ecosystem. We offer security reviews to projects built on EVM compatible chains and StarkNet. We have expertise in multiple areas of the Ethereum ecosystem, including protocol design, smart contracts (written in Solidity and Cairo), MEV, etc. We develop some of the most used tools on Starknet and one of the most used Ethereum clients. Learn more about us at https://nethermind.io.

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.