Security Review Report
NM-0741 BlueGoldOne

NETHERMIND
SECURITY

(December 09, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for BlueGoldOne's StandardGoldCoin Token contract.

Standard Gold Coin is a digitized commodity token designed to track the value of physical gold (1 gram per token). The system is a UUPS upgradeable ERC20 that implements: a unique issuance and redemption workflow to verify off-chain backing, a 0.02% transfer fee mechanism, regulatory blacklisting capabilities, emergency pause functionality, and **EIP2612 Permit** features.

The audit comprises 161 lines of the Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools.

Along this document, we report 4 points of attention, where one is classified as `Medium` and, three are classified as `Informational` 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 presents the system overview. 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 test suite evaluation and automated tools used. Section 8 concludes the document.

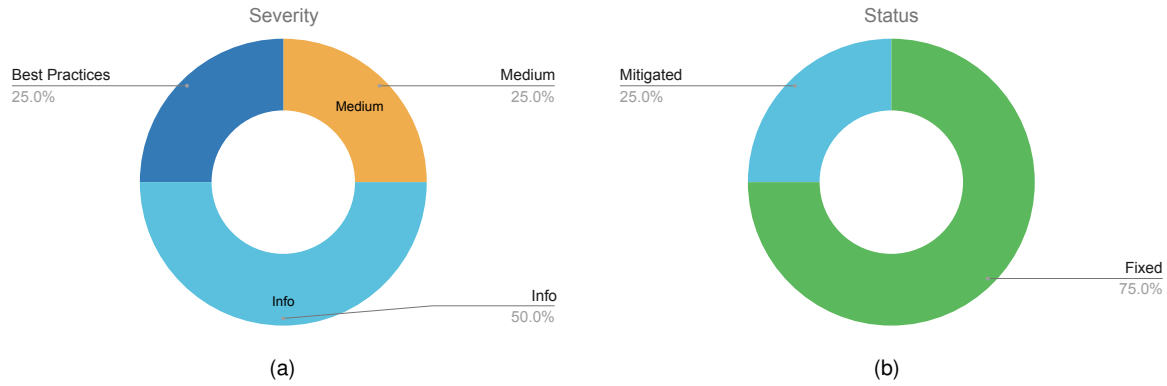

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (1), **Low** (0), **Undetermined** (0), **Informational** (2), **Best Practices** (1). **Distribution of status: Fixed** (3), **Acknowledged** (0), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	December 04, 2025
Final Report	December 09, 2025
Initial Commit	612f160
Final Commit	bbd7923
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/StandardGoldCoin.sol	161	295	183.2%	63	519
	Total	**161**	**295**	**183.2%**	**63**	**519**

3 Summary of Issues

	Finding	Severity	Update
1	Blacklisted spenders can still execute transfers via `transferFrom(...)`	Medium	Fixed
2	The EOA check in `initialize(...)` can be bypassed by contracts during construction	Info	Mitigated
3	`supportsInterface` returns false for ERC20, Metadata, and Permit interfaces	Info	Fixed
4	The implementation contract does not call `_disableInitializers(...)` in the constructor	Best Practices	Fixed

4 System Overview

The Standard Gold Coin (SGC) protocol implements a digitized commodity system where each SGC token represents one gram of physical gold. The system is architected as a UUPS Upgradeable ERC20 token that integrates regulatory compliance features, a transaction fee mechanism, and a strict issuance model to maintain a 1:1 correlation with off-chain gold reserves.

The protocol consists of a single primary implementation contract, `StandardGoldCoin`, which utilizes OpenZeppelin's upgradeable contracts library for core standards including `ERC20Permit`, `AccessControl`, and `Pausable`.

4.1 Issuance and Redemption Logic

The system distinguishes itself from standard ERC20 tokens by enforcing a strict "Declared Backing" model. The total supply is managed through restricted roles to ensure on-chain tokens match off-chain physical assets.

- **Minting**: The `mintWithIssuance` function allows the `MINTER_ROLE` to create tokens. Crucially, this operation requires a unique `issuanceId`. This ID links the on-chain minting event to a specific off-chain gold deposit, preventing duplicate processing of the same physical asset. The global `totalDeclaredBacking` counter is incremented upon success.
- **Redemption**: Users wishing to redeem tokens for physical gold undergo a burning process via the `burnForRedemption` function (executed by the `BURNER_ROLE`). Similar to minting, this requires a unique `redemptionId`. This ensures the `totalDeclaredBacking` is decremented correctly and provides an immutable record of the redemption. Users may also burn their own tokens voluntarily via the public `burn` function.

4.2 Transfer and Fee Mechanism

The protocol overrides the standard `_update` function to inject custom logic into every token transfer.

- **Transaction Fees**: A fee of 2 basis points (0.02%) is applied to standard transfers. This fee is deducted from the transfer amount and sent to a designated `feeCollector`.
- **Fee Exemptions**: Mints and burns are inherently exempt from fees. Additionally, transfers where the sender or recipient is the `feeCollector` are processed without fees to prevent circular charging.
- **Collector Constraints**: The protocol enforces a security constraint requiring the `feeCollector` to be an Externally Owned Account (EOA). The system explicitly checks `code.length` to prevent contract addresses from receiving fees.

4.3 Security and Compliance Controls

To adhere to regulatory requirements and ensure system safety, SGC implements centralized control mechanisms comparable to major stablecoins.

- **Blacklisting**: A `BLACKLIST_MANAGER_ROLE` can freeze specific addresses via `addBlackList` and `removeBlackList`. Blacklisted addresses are strictly prohibited from sending or receiving tokens. The implementation follows the standard `isBlackListed` interface for compatibility with external compliance tools.
- **Emergency Pause**: The `PAUSER_ROLE` can trigger a global pause, halting all token transfers, minting, and burning operations in the event of a security breach or system migration.
- **Access Control**: The system utilizes granular role-based permissions. Distinct roles are assigned for upgrading the contract (`UPGRADER_ROLE`), managing fees (`DEFAULT_ADMIN_ROLE`), and operational tasks (Minting/Burning).

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 [Medium] Blacklisted spenders can still execute transfers via `transferFrom(...)`

File(s): src/StandardGoldCoin.sol

Description: The `StandardGoldCoin` contract implements a blacklist mechanism designed to freeze malicious accounts and prevent them from participating in transactions. This restriction is enforced by overriding the `_update(...)` function to check both the sender (`from`) and the recipient (`to`).

However, the contract inherits the standard ERC20 `transferFrom(...)` logic without overriding it or `_spendAllowance(...)` to validate the status of the `spender`.

In a `transferFrom(...)` operation, the `msg.sender` is the `spender`. Since `_update(...)` only verifies the token owner and the recipient, a blacklisted address acting as a `spender` can still move funds on behalf of other users. This bypasses the intent of the blacklist, which is to isolate the malicious account from the protocol.

```
1
2  // @audit The _spendAllowance function doesn't check if the spender is blacklisted or not
3  function _update(address from, address to, uint256 value) internal override whenNotPaused {
4      // Enforce blacklist (skip for mint/burn)
5      if (from != address(0)) require(!_blacklisted[from], "SGC: sender blacklisted");
6      if (to != address(0)) require(!_blacklisted[to], "SGC: recipient blacklisted");
7
8      // ...
9  }
```

Recommendation(s): Consider overriding the `_spendAllowance(...)` function to explicitly require that `msg.sender` (the spender) is not blacklisted.

Status: Fixed

Update from the client: Fixed in 099e70c

6.2 [Info] The EOA check in `initialize(...)` can be bypassed by contracts during construction

File(s): /src/StandardGoldCoin.sol

Description: The `initialize(...)` function attempts to distinguish whether the provided `admin` address is an Externally Owned Account (EOA) or a smart contract to correctly assign the `feeCollector`. The protocol explicitly intends for the `feeCollector` to be `address(0)` if the admin is a contract to avoid assigning a contract as a fee collector.

However, the logic relies on `admin.code.length == 0` to verify EOA status. In Solidity, a smart contract returns a code size of `0` while it is being executed inside its own `constructor`.

This creates an edge case where a smart contract could call `initialize(...)` during its construction phase. The check would falsely identify it as an EOA, resulting in the contract address being assigned as the `feeCollector`, contrary to the protocol's intended design constraints.

```
1      function initialize(address admin) public initializer {
2          // ...
3          // @audit-issue Returns 0 if admin is a contract currently being constructed
4          if (admin.code.length == 0) {
5              feeCollector = admin;
6              emit FeeCollectorChanged(address(0), admin);
7          } else {
8              feeCollector = address(0);
9              emit FeeCollectorChanged(address(0), address(0));
10         }
11     }
```

Recommendation(s): Consider documenting this behavior to ensure operational teams are aware of it.

Status: Mitigated

Update from the client: Fixed in 099e70c

6.3 [Info] `supportsInterface` returns false for ERC20, Metadata, and Permit interfaces

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

Description: The contract overrides `supportsInterface` to ensure compatibility with `AccessControlUpgradeable`. However, the implementation simply delegates to `super.supportsInterface`.

Since OpenZeppelin's `ERC20Upgradeable` and `ERC20PermitUpgradeable` do not implement ERC165 by default, the `super` call chain does not account for them. Consequently, the contract reports that it does not support the following standard interfaces, despite fully implementing them `IERC20`, `IERC20Metadata`, and `IERC20Permit`.

This creates a discrepancy where the contract functions as an ERC20/Permit token but denies supporting these interfaces.

```
    // ...
    function supportsInterface(bytes4 interfaceId) public view override(AccessControlUpgradeable) returns (bool) {
        // @audit Returns false for IERC20, IERC20Metadata, and IERC20Permit
        return super.supportsInterface(interfaceId);
    }
```

Recommendation(s): Consider manually checking for these interface IDs to provide accurate contract information.

Status: Fixed

Update from the client: Fixed in 099e70c

Update from the Nethermind Security Team: You don't need to manually calculate interface ID. You can do something like:

```
return interfaceId == type(IERC20).interfaceId ||
    interfaceId == type(IERC20Metadata).interfaceId ||
    interfaceId == type(IERC20Permit).interfaceId ||
    super.supportsInterface(interfaceId);
```

Update from the client: Fixed in bbd7923

6.4 [Best Practice] The implementation contract does not call `_disableInitializers(...)` in the constructor

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

Description: The `StandardGoldCoin` contract is designed as an upgradeable contract that inherits from `Initializable` and `UUPSUpgradeable`. It relies on the `initialize(...)` function to set up the initial state, such as granting the `DEFAULT_ADMIN_ROLE` and other roles to the admin address.

However, the contract does not implement a constructor that calls `_disableInitializers()`.

This means that the logic contract (the implementation) can be initialized by any third party after deployment. According to OpenZeppelin's best practices for upgradeable contracts, the implementation contract should be "locked" in the constructor to prevent it from being used or initialized directly. Leaving it open can allow an attacker to claim ownership of the implementation contract.

```
contract StandardGoldCoin is
    Initializable,
    ERC20Upgradeable,
    ERC20PermitUpgradeable,
    AccessControlUpgradeable,
    PausableUpgradeable,
    UUPSUpgradeable
{
    // ...
    // @audit-issue The contract lacks a constructor calling _disableInitializers().

    //..
}
```

Recommendation(s): Consider adding a constructor to the contract that calls `_disableInitializers()`. This ensures that the implementation contract is locked upon deployment and cannot be initialized by an attacker later on.

Status: Fixed

Update from the client: Fixed in 099e70c

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 BlueGoldOne's documentation

The documentation for the BlueGoldOne protocol was provided through Natspec. Moreover, the team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
> forge test

Ran 20 tests for test/StandardGoldCoin.t.sol:StandardGoldCoinMintBlacklistTest
[PASS] testBlacklistedRecipientCannotReceive() (gas: 147509)
[PASS] testBlacklistedSenderCannotTransfer() (gas: 145290)
[PASS] testBurnForRedemptionBackingUnderflow() (gas: 52813)
[PASS] testBurnForRedemptionDoubleIdFails() (gas: 150259)
[PASS] testBurnForRedemptionZero() (gas: 15781)
[PASS] testMintLarge() (gas: 115641)
[PASS] testMintRevertsIssuanceUsed() (gas: 115885)
[PASS] testMintRevertsZeroAmount() (gas: 15804)
[PASS] testNoFeeOnBurn() (gas: 120576)
[PASS] testNoFeeOnMint() (gas: 114519)
[PASS] testOnlyBlacklistManagerCanAdd() (gas: 77211)
[PASS] testOnlyBlacklistManagerCanRemove() (gas: 137244)
[PASS] testOnlyBurnerCanBurnForRedemption() (gas: 13679)
[PASS] testOnlyMinterCanMint() (gas: 13702)
[PASS] testOnlyPauserCanPause() (gas: 13418)
[PASS] testPauseBlocksTransfers() (gas: 144559)
[PASS] testRemoveFromBlacklistAllowsTransfer() (gas: 188411)
[PASS] testTransferWithFee() (gas: 174689)
[PASS] testUnpauseRestoresTransfers() (gas: 184213)
[PASS] testUserBurn() (gas: 120590)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 6.68ms (8.48ms CPU time)

Ran 1 test suite in 87.48ms (6.68ms CPU time): 20 tests passed, 0 failed, 0 skipped (20 total tests)
```

8.2 Automated Tools

8.2.1 AuditAgent

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

9 About Nethermind

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

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

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

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

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

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to 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.

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.