Security Review Report
NM-0083 PIMLICO

NETHERMIND

(May 2, 2023)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 3

4 Risk Rating Methodology 3

5 Issues 4

5.1 [High] `tokenAmount` calculation can be incorrect depending on token decimals . . . 4

5.2 [Low] Chainlink's `latestRoundData` might return stale or incorrect results . . . 5

5.3 [Low] Missing input validation for `priceUpdateThreshold` . . . 6

5.4 [Best Practice] Missing use of `SafeERC20` . . . 6

6 Documentation Evaluation 7

7 Test Suite Evaluation 8

7.1 Contracts Compilation Output . . . 8

7.2 Tests Output . . . 8

7.3 Code Coverage . . . 8

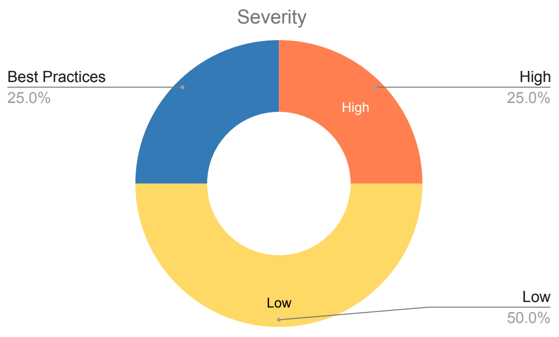
8 About Nethermind 9

1 Executive Summary

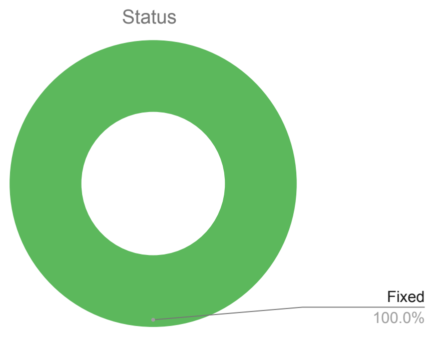
This document presents the security review performed by Nethermind on the Pimlico ERC20 Paymaster contract. **Pimlico** is a company that specializes in providing infrastructure for the EIP-4337 account abstraction standard, which aims to improve the efficiency and flexibility of account management on blockchain networks. This standard allows for easier and more secure management of accounts, by implementing account logic into a smart contract rather than relying on an EOA. Pimlico's focus is on building infrastructure for the EIP-4337 standard, particularly for paymasters and bundlers, while also developing an API that allows developers to easily integrate with account abstraction and sponsored transactions.

The audit was performed using manual analysis of the codebase as well as automated analysis tools. Along this document, we report 4 points of attention, where 1 is classified as *High*, 2 are classified as *Low* and 1 is classified as *Best Practices*. The issues are summarized in Fig. 1.

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



(a)



(b)

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (0), **Best Practices** (1).
Distribution of status: Fixed (4), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr. 24, 2023
Response from Client	Apr. 30, 2023
Final Report	May. 2, 2023
Methods	Manual Review, Automated Analysis
Repository	pimlicolabs/erc20-paymaster-contracts
Commit Hash (Initial Audit)	1d32643ff9b9dbd17c97d86bb396ae10bf62f085
Documentation	README.md, Pimlico Documentation Site, EIP-4337
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/PimlicoERC20Paymaster.sol	86	32	37.2%	13	131
	Total	**86**	**32**	**37.2%**	**13**	**131**

3 Summary of Issues

	Finding	Severity	Update
1	`tokenAmount` calculation can be incorrect depending on token decimals	High	Fixed
2	Chainlink's `latestRoundData` might return stale or incorrect results	Low	Fixed
3	Missing input validation for `priceUpdateThreshold`	Low	Fixed
4	Missing use of `SafeERC20`	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, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

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

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

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

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

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

5 Issues

5.1 [High] `tokenAmount` calculation can be incorrect depending on token decimals

File(s): `PimlicoERC20Paymaster.sol`

Description: In the function `_validatePaymasterUserOp(...)` the paymaster contract will `transferFrom` ERC20 tokens from the user as payment for the transaction execution. The contract employs a Chainlink oracle to determine the price of ETH and calculate the correspending amount of ERC20 tokens required. However, the calculation does not take into account the number of decimals in the token, which can lead to users paying more or less than expected.

```
/////////////////////////////////////////////////////////
// @audit Wrong formula, not accounted for the decimals
/////////////////////////////////////////////////////////
uint256 tokenAmount = (
    requiredPreFund + (REFUND_POSTOP_COST) * userOp.maxFeePerGas
) * priceMarkup / cachedPrice;
```

Since `priceMarkup` has 6 decimals, this calculation will only output a correct result when the `token` is also 6 decimals. If the `token` decimal has any other value, it will affect the resulting `tokenAmount` value which can lead to one of the two situations:

- When `token` decimals is less than 6 decimals, the user will pay more tokens than expected;
- When `token` decimals is more than 6 decimals, the user will pay less tokens than expected;

The following is a demonstration of this behavior using the DAI token, which is 18 decimals:

1. The Chainlink feed DAI/ETH returns `latestAnswer = 518927045882705`. This means that 1 DAI is equal to 518927045882705 Wei, or 0.0005189 Ether;
2. To make calculations simple, we assume `REFUND_POSTOP_COST` to be zero, and `priceMarkup` to be 100%;
3. For a `requiredPreFund` of 1 Ether, the required amount of ERC20 tokens can be calculated as follows:;

```
    tokenAmount = 1e18 * 1e6 / 518927045882705 = 1927053153;
```

4. Since DAI has 18 decimal places, the resulting `tokenAmount` actually represents an extremely small USD value even though the user was expected to pay 1 Ether worth of DAI. In this case, the user only has to pay 0.000000001927053153 DAI for 1 Ether of execution gas cost;

Recommendation: Correct the formula by taking into account the decimals of the token to ensure pricing is correct.

Status: Fixed

Update from the client: Supporting non 6 decimal tokens by calculating the tokenAmount with token's decimal which is given on constructor. Fixed in PR: https://github.com/pimlicolabs/erc20-paymaster-contracts/pull/4

5.2 [Low] Chainlink's `latestRoundData` might return stale or incorrect results

File(s): `PimlicoERC20Paymaster.sol`

Description: The functions `updatePrice(...)` and `_postOp(...)` both make a call to a Chainlink oracle to receive the `latestRoundData`. If there are any problems with Chainlink starting a new round and finding consensus on the new price for the oracle (e.g. Chainlink nodes abandon the oracle, chain congestion, vulnerability/attack on the Chainlink system), the `PimlicoERC20Paymaster` contract may continue using stale or incorrect data. This could lead to users paying more or less ERC20 tokens to cover their gas costs than expected.

Recommendation: The function `updatePrice(...)` can prevent updates with potentially stale or incorrect data by adding the following checks:

```
(
    uint80 roundId,
    int256 answer,
    ,
    uint256 updatedAt,
    uint80 answeredInRound
) = oracle.latestRoundData();

require(answer > 0, "Chainlink price <= 0");
require(updatedAt != 0, "Incomplete round");
require(answeredInRound >= roundId, "Stale price");
```

It should be noted that this recommendation could have some side-effects if applied to `_postOp(...)`, since a reverting post-operation could affect reputation.

One potential solution for `_postOp(...)` is to accept the potential reputation risk, as the effects of accepting mispriced assets could lead to loss of funds, whereas reputation can be recovered. Alternatively, to avoid reputation loss the function could refer to a backup price via Uniswap if the Chainlink data is stale.

Status: Fixed

Update from the client: Using 2 days as indicator for checking if oracle is not getting maintained. Also we use 2 oracles instead of one. EG. use USDC/USD oracle and ETH/USD oracle to calculate the USD/ETH ratio. Fixed in PR: https://github.com/pimlicolabs/erc20-paymaster-contracts/pull/5

5.3 [Low] Missing input validation for `priceUpdateThreshold`

File(s): `PimlicoERC20Paymaster.sol`

Description: The function `updateConfig(...)` is missing input validation for the `priceUpdateThreshold` argument. If this is set to a very high value (e.g. above `priceDenominator`), it may lead to an overflow in the function `_postOp(...)` when calculating whether the price should be updated. This overflow would not revert since it is located within an `unchecked` block. This could cause price updates to behave incorrectly.

```
function updateConfig(uint32 _priceMarkup, uint32 _updateThreshold) external onlyOwner {
    require(_priceMarkup <= 120e4, "price markup too high");
    require(_priceMarkup >= 1e6, "price markeup too low");
    priceMarkup = _priceMarkup;

    ////////////////////////////////////////////////////////
    // @audit no input validation for priceUpdateThreshold
    ////////////////////////////////////////////////////////
    priceUpdateThreshold = _updateThreshold;
    emit ConfigUpdated(_priceMarkup, _updateThreshold);
}
```

Recommendation(s): Consider adding input validation for `priceUpdateThreshold`.

Status: Fixed

Update from the client: Fixed in PR: https://github.com/pimlicolabs/erc20-paymaster-contracts/pull/3

5.4 [Best Practice] Missing use of `SafeERC20`

File(s): `PimlicoERC20Paymaster.sol`

Description: When transferring ERC20 tokens, the functions `transfer(...)` and `transferFrom(...)` may not account for some edge cases in token behavior. An example of such a behavior is ZRX, which does not revert on failure but instead returns `false`. This could lead to silent transfer failures, where the paymaster contract would continue to execute even though no tokens have been transferred.

Recommendation(s): Consider using `safeTransfer(...)` and `safeTransferFrom(...)` from the OpenZeppelin SafeERC20 library.

Status: Fixed

Update from the client: As addressed in audit report, enabling any token with same code base will be easier for us to maintain the project. Thus using safeTransfer, using modified version of solady's SafeTransferLib Library has been modified because Entrypoint reverts on short errors. Fixed in PR: https://github.com/pimlicolabs/erc20-paymaster-contracts/pull/2

6 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 plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. 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 that describes the design and technical details of the smart contract. 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.

Three sources of documentation were used during the audit, (a) README, (b) EIP-4337 Abstract, (c) NatSpec and inline comments. A summary on each of the sources is presented below:

README: The README is highly detailed, containing a general overview as well as a detailed list of each function and event. A development setup section is also present, explaining how to prepare the environment as well as build and execute tests.

EIP-4337 Abstract: Although this is not official documentation provided by Pimlico, this specification has served to be very useful throughout the audit. It contains the entire EIP-4337 standard in full detail.

NatSpec and inline comments: The in-line documentation is high quality and covers every function. For each function its general purpose, arguments and returns and described. In-line comments are also present for areas in the code that need further explanations.

7 Test Suite Evaluation

7.1 Contracts Compilation Output

```
> forge build
[] Compiling...
[] Compiling 67 files with 0.8.19
[] Solc 0.8.19 finished in 3.39s
Compiler run successful (with warnings)
warning[3420]: Warning: Source file does not specify required compiler version! Consider adding "pragma solidity
 ↪  ^0.8.19;"
--> src/test/TestERC20.sol

# Warnings for the test file `PimlicoERC20Paymaster.t.sol` have been omitted.
```

7.2 Tests Output

```
> forge test
[] Compiling...
[] Compiling 2 files with 0.8.19
[] Solc 0.8.19 finished in 2.34s
Compiler run successful (with warnings)

Running 17 tests for test/foundry/PimlicoERC20Paymaster.t.sol:PimlicoERC20PaymasterTest
[PASS] testCall() (gas: 304955)
[PASS] testDeploy() (gas: 1471132)
[PASS] testERC20PaymasterFailNotUpdated() (gas: 307947)
[PASS] testERC20PaymasterFailWeirdCalldataLength() (gas: 317509)
[PASS] testERC20PaymasterPostOpFailed() (gas: 399720)
[PASS] testERC20PaymasterRefundAndGuaredToken() (gas: 400941)
[PASS] testERC20PaymasterRefundAndGuaredTokenFailTokenTooHigh() (gas: 318461)
[PASS] testERC20PaymasterRefundAndNoGuaredToken() (gas: 400179)
[PASS] testERC20PaymasterUpdatePriceDown() (gas: 408172)
[PASS] testERC20PaymasterUpdatePriceUp() (gas: 408132)
[PASS] testOwnershipTransfer() (gas: 21760)
[PASS] testUpdateConfigFailMarkupTooHigh(uint32,uint32) (runs: 256, : 19694, ~: 19878)
[PASS] testUpdateConfigFailMarkupTooLow(uint32,uint32) (runs: 256, : 19748, ~: 19842)
[PASS] testUpdateConfigSuccess(uint32,uint32) (runs: 256, : 27823, ~: 27716)
[PASS] testUpdatePrice(int192) (runs: 256, : 25846, ~: 25846)
[PASS] testWithdrawToken(uint256) (runs: 256, : 57384, ~: 58715)
[PASS] testWithdrawTokenFailNotOwner(uint256) (runs: 256, : 47342, ~: 49382)
Test result: ok. 17 passed; 0 failed; finished in 40.77ms
```

7.3 Code Coverage

```
> forge coverage
```

The relevant output is presented below.

```
| File                            | % Lines         | % Statements    | % Branches      | % Funcs         |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|
| src/PimlicoERC20Paymaster.sol   | 100.00% (31/31) | 100.00% (35/35) | 100.00% (18/18) | 100.00% (5/5)   |
| src/test/TestCounter.sol        | 16.67% (1/6)    | 12.50% (1/8)    | 100.00% (0/0)   | 25.00% (1/4)    |
| src/test/TestERC20.sol          | 100.00% (3/3)   | 100.00% (3/3)   | 100.00% (0/0)   | 100.00% (3/3)   |
| src/test/TestOracle.sol         | 66.67% (2/3)    | 66.67% (2/3)    | 100.00% (0/0)   | 66.67% (2/3)    |
| test/foundry/BytesLib.sol       | 0.00% (0/50)    | 0.00% (0/50)    | 0.00% (0/22)    | 0.00% (0/14)    |
| Total                           | 39.78% (37/93)  | 41.41% (41/99)  | 45.00% (18/40)  | 37.93% (11/29)  |
```

8 About Nethermind

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

Disclaimer