OpenZeppelin | security

Pimlico ERC20Paymaster Audit

PIMLICO

March 28, 2024

Table of Contents

Summary

Type	Account Abstraction
Timeline	From 2024-3-11 To 2024-3-14
Languages	Solidity

Total Issues	13 (12 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	3 (3 resolved)
Notes & Additional Information	8 (8 resolved)
Client Reported Issues	1 (0 resolved)

Scope

We audited the `ERC20Paymaster.sol` file present in the pimlicolabs/erc20-paymaster repository at commit 8e37933.

System Overview

The audited system is an EIP-4337-compliant paymaster that allows users to pay for their transactions in ERC-20 tokens instead of ETH. This paymaster has added two dimensions to its features: the ability to specify a maximum token amount to spend for a transaction and the ability to specify a "guarantor" who will pay for any lack of tokens from the user but will require the user to attempt to pay said guarantor the transaction cost. The guarantor needs special attention, however, as it can do things not generally allowed by ERC-4337. This could lead to the paymaster getting banned in the canonical bundler mempool.

The paymaster is intended to be used as a staked paymaster which allows it to read Chainlink prices in the operation validation phase where the maximum token cost is pre-charged. The token price, along with the pre-charged token amount and other data, is then passed as part of the `context` to the post-operation phase where actual execution costs are accounted for and settled. This architecture would not work if the paymaster were unstaked (or became unstaked).

Privileged Roles and Trust Assumptions

The system relies on Chainlink oracles to fetch accurate prices for the token at the validation stage. It also relies on the Solady library to perform token transfers. We have not audited either of those two systems and expect them to work as intended.

The paymaster allows a single "owner" to:

- Update the token price markup (intended to specify the profit margin of the paymaster)
- Withdraw any ERC-20 tokens held by the paymaster
- Withdraw any ETH deposited by the paymaster into the EIP-4337-specified `EntryPoint`
- Manage the paymaster's stake at the `EntryPoint` (i.e., add, unlock, or withdraw stake)

For this audit, we assume that the paymaster will always be staked, thereby allowing it to access shared storage during the validation phase as per EIP-4337's reputation rules.

Medium Severity

M-01 REFUND_POSTOP_COST Is Small for Some Tokens

`REFUND_POSTOP_COST` is a constant set to 30000. The actual gas cost of `PostOp` depends on the gas cost of a token transfer since it can perform up to two token transfers. This means that the gas cost will be higher for tokens with high transfer cost (like stETH or any other token with extra functionality). Using a token like stETH would cause a loss to the paymaster as it would be paying ETH for the whole cost of `postOp`, including the token rebasing, but only charging the user for 30000 gas. The gas cost could also be lower for a token with a very optimized transfer, resulting in a higher-than-needed fee paid by users. In addition, the gas cost of `PostOp` can vary depending on whether a guarantor is used or not.

Consider testing the gas cost for `PostOp` for each instance that uses a different token and setting `REFUND_POSTOP_COST` in the constructor according to those tests. If a token upgrades to a version that consumes a different amount of gas on transfer, the paymaster should be redeployed with a new `REFUND_POSTOP_COST` value. To prevent this from ever happening, consider adding a function for the owner to change `REFUND_POSTOP_COST` in case of a token upgrade. Also, consider charging a different `PostOp` cost depending on the chosen mode since transactions paid for by guarantors will consume more gas.

***Update:** Resolved in pull request #17 at commit dfc1637.*

Low Severity

L-01 Incompatible ERC-20 Tokens

The paymaster is not compatible with down-rebasing tokens and fee-on-transfer tokens. For the latter, the received amount can be smaller than the transferred amount, which is never checked. For down-rebasing tokens, a rebase can be activated during execution so the balance will be different in `PostOp` than in validation, causing a revert in case the refund fails (entailing a loss of the paymaster's reputation) or a loss of profit for the paymaster.

Consider making it explicit in the documentation that fee-on-transfer tokens and down-rebasing tokens are not supported.

Update: *Resolved at pull request #18 at commit 2888d16.*

L-02 Inaccurate Staleness Threshold

According to this comment, a two-day staleness threshold has been chosen to give a one-day-long extra buffer to the oracle's 24-hour max heartbeat. However, while USDC/USD has a heartbeat of 24 hours, DAI/USD and COMP/USD each have heartbeats of one hour.

Consider whether each contract should have the same, one-day staleness threshold. If so, consider having a constant equal to `1 day` and a variable set in the constructor equal to the max heartbeat. The threshold can then be set to `1 day + heartbeat`. If each paymaster should have different staleness thresholds, consider adding that as an immutable variable set at construction.

Update: *Resolved at pull request #22 at commit 853b9a2.*

L-03 Mode Parsing Accepts Malformed `paymasterData`

As per the documentation for how to specify mode one for the paymaster, `paymasterData` must be empty to do so. However, calling the paymaster with `0x00` as the first byte in `paymasterData` will get parsed by `_parsePaymasterAndData` as mode one.

Consider either documenting this as an additional option for specifying mode one or adding checks to ensure that this data is rejected.

Update: *Resolved at pull request #23 at commit f2013ff.*

Notes & Additional Information

N-01 Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease code clarity and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity file or when inheritance chains are long. Within `ERC20Paymaster.sol`, all of the imports are global imports and a few even import the same underlying interfaces.

Following the principle that clearer code is better code, consider using the named import syntax *(`import {A, B, C} from "X"`)* to explicitly declare which contracts are being imported.

Update: *Resolved at pull request #21 at commit a2698d5.*

N-02 Floating Pragma

It is often a good idea for pragma directives to be fixed to clearly identify which version of the compiler should be used with this contract. The ERC20Paymaster.sol file has the solidity ^0.8.23 floating pragma directive.

Consider setting a fixed pragma version.

Update: *Resolved at pull request #25 at commit 588b9e1.*

N-03 Revert Without Error on `calldata` Array Access

In `_validatePaymasterUserOp`, `paymasterConfig` is often sliced at indices which might cause the transaction to revert if the user has not formed their `paymasterData` correctly and the byte string is not as long as expected. While this should not pose a risk to the reputation of the contract, it will nonetheless revert without a discernable error.

To improve the readability of the codebase and make tests more understandable, consider adding specific checks and errors for malformed `paymasterData` bytes.

Update: *Resolved at pull request #26 at commit d759f72.*

N-04 Chainlink's `answeredInRound` Is Deprecated

In the `_fetchPrice` function, the Chainlink oracle is queried. One of the return values is `answeredInRound` which is used to determine if the returned price is stale. Chainlink's documentation states that this value is deprecated and we confirmed with their team that "developers should avoid using `answeredInRound` entirely now".

Consider removing the usage of this value in this function.

Update: *Resolved at pull request #22 at commit 853b9a2.*

N-05 Inaccurate Error Name

The `OraclePriceZero` error is utilized when the oracle price is non-positive.

Consider renaming this error to `OraclePriceNotPositive`

Update: *Resolved at pull request #22 at commit 853b9a2.*

N-06 Mode Check Could Be Simplified

In `_validatePaymasterOp`, the `paymasterAndData` value is parsed and the specified mode is checked by using a bit mask. The intent of the code would be much clearer here if the mode were checked to be less than five and it would also save about 25 gas.

Consider rewriting this check using the less-than (<) operator.

Update: *Resolved at pull request #27 at commit 94f62a5.*

N-07 Parameter Description Is Not Clear

In the docstring for `maxCost` in the `_validatePaymasterUserOp` function, `maxCost` is described as being "the amount of tokens required for pre-funding." However, the tokens being referred to here are native tokens and could be misconstrued given that the contract deals with ERC-20 tokens.

Consider modifying the docstring to specify that the `maxCost` is in native tokens.

Update: *Resolved at pull request #28 at commit db15a0a.*

N-08 Unnecessary Casting

When `_validatePaymasterOp` wants to run different code for different modes, it compares the mode to casted integers. However, this casting is unnecessary as the compiler will ensure that the different values are the correct type to be compared against `mode`.

Consider removing the `uint8` casting for these three mode comparisons.

Update: *Resolved at pull request #29 at commit 0f74e9e.*

Client Reported

CR-01 Paymaster DOS Attack Vector

Since `ERC20Paymaster` is intended to be staked, it is able to, and indeed does, query the price oracle during the operation validation phase. This creates a small window between bundle validation and acceptance by the `EntryPoint` where the oracle price could change just enough so that the paymaster reverts on-chain where it did not in simulation. The code could fail if the price dropped enough so that pre-charged tokens no longer cover the actual token cost of execution.

This issue is an accepted risk. The Pimlico team stated:

> An attack utilizing this vector is not scalable. There would not be a reliable way to pull it off.

Update: *Acknowledged, not resolved.*

Conclusion

Pimlico's `ERC20Paymaster` contract allows users to pay for operations using ERC-20 tokens. This is designed to work within EIP-4337's broader specification of account abstraction.

The audit yielded one medium-severity issue along with various lower-severity issues and best practice recommendations. The most pertinent issue was related to taking special care with the `postOp` gas estimations for different modes and tokens. The paymaster implementation was mature and well-documented which we were very appreciative of.

The Pimlico team was also very responsive throughout the audit period and consistently engaged with our queries. We commend for them their commitment to making this audit effective.