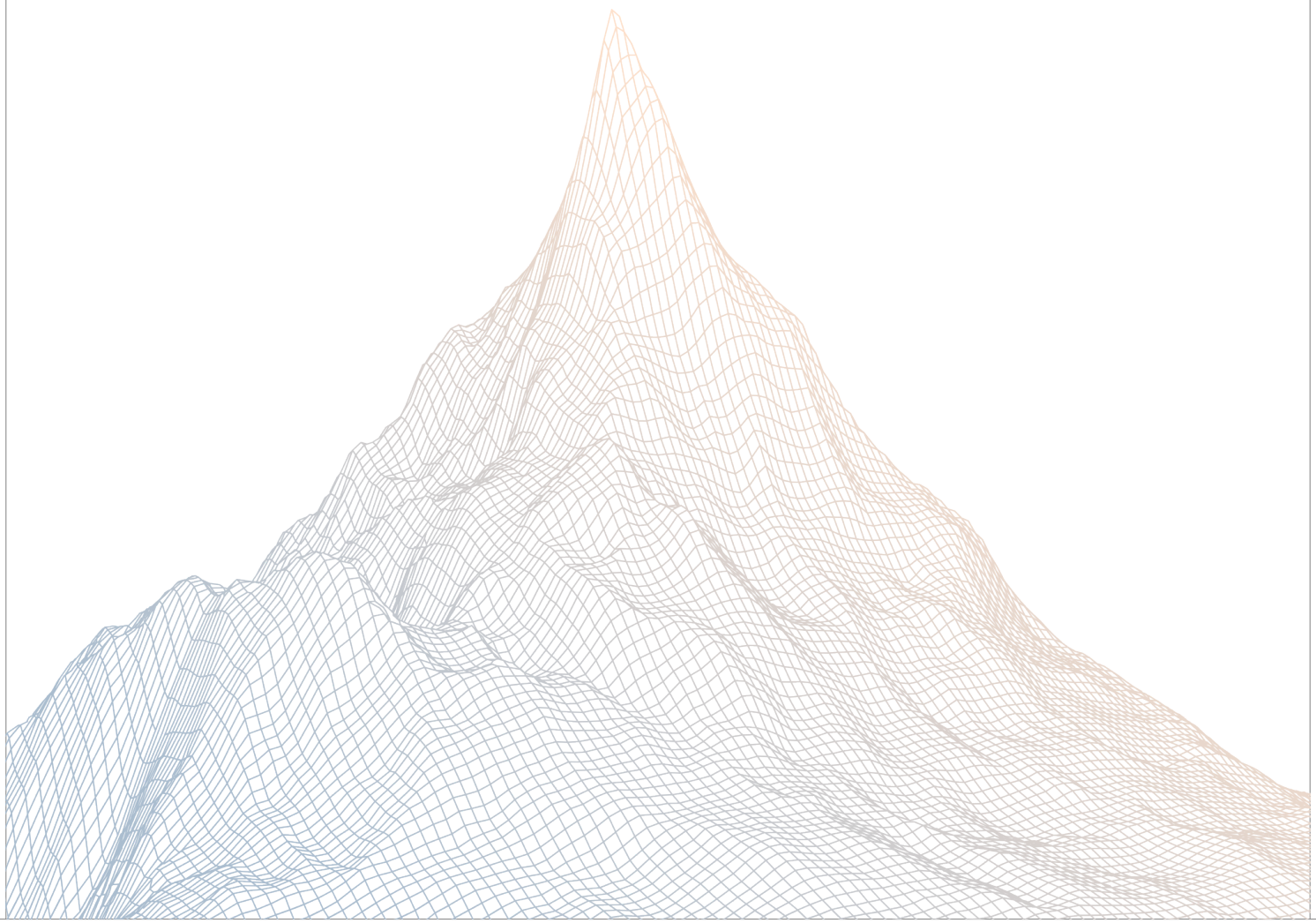
Zenith

Biconomy

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: April 15th to April 17th, 2025

AUDITED BY: Riley Holterhus
Said

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Biconomy MEE

Biconomy Modular Execution Environment is a permissionless network which can provide credible execution for a variety of offchain and onchain instructions - contained within the Supertransaction data model. The audited smart contracts are the on-chain component for Biconomy MEE.

2.2 Scope

The engagement involved a review of the following targets:

Target	mee-contracts
Repository	https://github.com/bcnmy/mee-contracts
Commit Hash	`fc7319ba3714ac21534004225e412c198650f1a2`
Files	`Changes in PR-31`

2.3 Audit Timeline

April 15, 2025	Audit start
April 17, 2025	Audit end
April 22, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	2
Informational	3
Total Issues	**5**

3

Findings Summary

ID	Description	Status
L-1	_isContract check may not work as intended once EIP-7702 is implemented	Resolved
L-2	_handleFixedPremium may revert due to an underflow.	Resolved
I-1	Precision loss when calculating the refund for percentage premium is unfavorable to the MEE node	Acknowledged
I-2	postOp() executes before paymaster receives ETH refund	Resolved
I-3	Leftover references to implied cost premium	Resolved

4

Findings

4.1 Low Risk

A total of 2 low risk findings were identified.

[L-1] `_isContract` check may not work as intended once EIP-7702 is implemented

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- K1MeeValidator.sol#L94
- K1MeeValidator.sol#L111-L115
- K1MeeValidator.sol#L312-L318

Description:

When `K1MeeValidator` is installed or transferring ownership of the smart accounts, it will ensure that the `newOwner` is not a contract.

```
function _isContract(address account) private view returns (bool) {
    uint256 size;
    assembly {
        size := extcodesize(account)
    }
    return size > 0;
}
```

However, once EIP-7702 is implemented and an EOA delegates to a contract, `EXTCODESIZE` will return 23, according to the EIP-7702 specification. This means that if this module is intended to coexist with EIP-7702-compatible EOAs, the check may cause unexpected reverts.

Recommendations:

Consider adjusting `_isContract` to return true only if the code size is greater than 0 and not equal to 23.

Biconomy: Resolved with @d63dfdd5e1...

Zenith: Verified.

[L-2] `_handleFixedPremium` may revert due to an underflow.

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BaseNodePaymaster.sol#L225

Description:

When handling fixed premium refund calculation, the refund is calculated by subtracting `actualGasCost` from `maxGasCost`. However, if `actualGasCost` is greater than `maxGasCost`, the operation will revert. In contrast, the percentage premium refund calculation inside `_handlePercentagePremium` avoids reverting by ensuring that the `refund` is only calculated when `costWithPremium` is less than `maxCostWithPremium`.

```
    function _handleFixedPremium(
        bytes calldata context,
        uint256 actualGasCost,
        uint256 actualUserOpFeePerGas
    ) internal pure returns (address refundReceiver, uint256 refund) {

        uint256 maxGasCost;
        uint256 postOpGasLimit;

        assembly {
            refundReceiver := shr(96, calldataload(context.offset))
            maxGasCost := calldataload(add(context.offset, 0x14))
            postOpGasLimit := calldataload(add(context.offset, 0x34))
        }

        // account for postOpGas
        actualGasCost += postOpGasLimit * actualUserOpFeePerGas;

        // when premium is fixed, payment by superTxn sponsor is maxGasCost +
    fixedPremium
        // so we refund just the gas difference, while fixedPremium is going
    to the MEE Node
>>>     refund = maxGasCost - actualGasCost;
    }
```

Recommendations:

Consider calculating `refund` only when `maxGasCost` is greater than `actualGasCost`.

Biconomy: Resolved with PR-36

Zenith: Verified.

4.2 Informational

A total of 3 informational findings were identified.

[I-1] Precision loss when calculating the `refund` for percentage premium is unfavorable to the MEE node

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BaseNodePaymaster.sol#L252-L262

Description:

When `_handlePercentagePremium` is executed, it first calculates `costWithPremium`, then `maxCostWithPremium`, and finally computes the `refund` by subtracting `costWithPremium` from `maxCostWithPremium`.

```
    function _handlePercentagePremium(
        bytes calldata context,
        uint256 actualGasCost,
        uint256 actualUserOpFeePerGas
    ) internal pure returns (address refundReceiver, uint256 refund) {

        uint192 premiumPercentage;
        uint256 maxGasCost;
        uint256 postOpGasLimit;

        assembly {
            refundReceiver := shr(96, calldataload(context.offset))
            premiumPercentage := shr(64, calldataload(add(context.offset,
0x14)))
            maxGasCost := calldataload(add(context.offset, 0x2c))
            postOpGasLimit := calldataload(add(context.offset, 0x4c))
        }

        // account for postOpGas
```

```
        actualGasCost += postOpGasLimit * actualUserOpFeePerGas;

        // we do not need to account for the penalty here because it goes to
    the beneficiary
        // which is the MEE Node itself, so we do not have to charge user for
    the penalty

        // account for MEE Node premium
>>>     uint256 costWithPremium = _applyPercentagePremium(actualGasCost,
    premiumPercentage);

        // as MEE_NODE charges user with the premium
        uint256 maxCostWithPremium = _applyPercentagePremium(maxGasCost,
    premiumPercentage);

        // We do not check for the case, when costWithPremium > maxCost
        // maxCost charged by the MEE Node should include the premium
        // if this is done, costWithPremium can never be > maxCost
        if (costWithPremium < maxCostWithPremium) {
>>>         refund = maxCostWithPremium - costWithPremium;
        }
    }
```

Precision loss when calculating `costWithPremium` using `_applyPercentagePremium` will be included in the `refund`.

```
function _applyPercentagePremium(uint256 amount, uint256 premiumPercentage)
    internal pure returns (uint256) {
    return amount * (PREMIUM_CALCULATION_BASE + premiumPercentage)
    / PREMIUM_CALCULATION_BASE;
}
```

Recommendations:

Consider rounding up the `costWithPremium` calculation

Biconomy: Acknowledged, MEE Node has multiple mechanisms to mitigate this, and in most cases it slightly overcharges the superTxn sponsor.

[I-2] `postOp()` executes before paymaster receives ETH refund

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- BaseNodePaymaster.sol#L163-L166

Description:

The `NodePaymaster` pays the max gas cost for a userOp upfront, and later calculates the ETH refund amount in the `postOp()` function. Depending on the refund type, this amount is forwarded to a recipient using `entryPoint.withdrawTo()`.

It's worth noting that when `postOp()` is called, the paymaster has not yet received the ETH refund. The increment to the paymaster's ETH balance in the entrypoint occurs after the `postOp()` completes. As a result, any call to `withdrawTo()` during `postOp()` is using the paymaster's existing balance.

This isn't necessarily a problem, since the paymaster needs to maintain an ETH balance in the entrypoint anyway. However this behavior could lead to reverts when the balance is low and may be worth documenting.

Recommendations:

Consider documenting this behavior in the code. For example:

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost,
    uint256 actualUserOpFeePerGas)
    internal
    virtual
    override
{
    //...
    // send refund to the superTxn sponsor
    if (refund > 0) {

    // Note: At this point the paymaster hasn't received the refund yet,
          so this withdrawTo() is
```

```
        // using the paymaster's existing balance. The paymaster's deposit in
            the entrypoint will be
        // incremented after postOp() concludes.
        entryPoint.withdrawTo(payable(refundReceiver), refund);
    }
}
```

Biconomy: Added comment in PR-37.

Zenith: Verified.

[I-3] Leftover references to implied cost premium

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- BaseNodePaymaster.sol
- NodePaymaster.sol
- Constants.sol#L20

Description:

There are comments in the code referring to the "implied cost" premium type. These appear above `_validate()` and `_postOp()` in the `BaseNodePaymaster`, and above `_validatePaymasterUserOp()` in the `NodePaymaster`.

The "implied cost" concept seems to be from an earlier version of the code and is no longer relevant. Also note that the constant variable `NODE_PM_PREMIUM_IMPLIED` in `Constants.sol` is unused in the codebase.

Recommendations:

Consider removing the outdated comments and deleting the unused `NODE_PM_PREMIUM_IMPLIED` constant.

Biconomy: Addressed in PR-37.

Zenith: Verified.