code4rena

# Biconomy

## Smart Contract Security Assessment

Version 1.0

Audit dates: Feb 03 — Feb 07, 2025

Audited by: Riley Holterhus
cccz

# 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](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

Biconomy Modular Execution Environment is a permisionless 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

| | |
|---|---|
| Repository | bcnmy/mee-contracts |
| Commit Hash | 5111e93dcd524a8c959eb2c62d266dbbf272fa5d |

### 2.3 Audit Timeline

| DATE | EVENT |
|---|---|
| Feb 03, 2025 | Audit start |
| Feb 07, 2025 | Audit end |
| Feb 16, 2025 | Report published |

### 2.4 Issues Found

| SEVERITY | COUNT |
|---|---|
| Critical Risk | 0 |
| High Risk | 0 |
| Medium Risk | 2 |
| Low Risk | 2 |
| Informational | 9 |
| Total Issues | 13 |

## 3. Findings Summary

| ID | DESCRIPTION | STATUS |
|---|---|---|
| M-1 | Hardcoded `RLP_ENCODED_R_S_BYTE_SIZE` can be incorrect | Resolved |
| M-2 | Potential underflow in `_calculateRefund()` | Resolved |

| | | |
|---|---|---|
| L-1 | Signatures might unintentionally match prefixes | Acknowledged |
| L-2 | Calls to `permit()` may be frontran | Resolved |
| I-1 | Phishing risks and unintended smart account actions | Acknowledged |
| I-2 | Appending extra calldata bytes may not always be possible | Acknowledged |
| I-3 | Padding extra data in signatures can increase gas costs | Acknowledged |
| I-4 | `TxValidatorLib` does not support type 1 transactions | Acknowledged |
| I-5 | `NodePaymaster` is not compatible with ERC4337 mempool | Acknowledged |
| I-6 | `_fillSafeSenders()` input size is not enforced | Resolved |
| I-7 | `_adjustV()` logic is unnecessary in `PermitValidatorLib` | Resolved |
| I-8 | `onUninstall()` does not clear `_safeSenders` storage | Resolved |
| I-9 | `tryRecover()` considerations | Acknowledged |

# 4. Findings

## 4.1 Medium Risk

A total of 2 medium risk findings were identified.

### [M-1] Hardcoded `RLP_ENCODED_R_S_BYTE_SIZE` can be incorrect

| Severity: Medium | Status: Resolved |
| --- | --- |

Target

- TxValidatorLib.sol#L244-L247

Severity:

- Impact: Medium
- Likelihood: Medium

**Description:** In the `TxValidatorLib`, the signature is expected to contain an RLP-encoded transaction signed by the user's EOA, with the last 32 bytes of the transaction's calldata representing a superTx merkle root.

In order to verify that the RLP-encoded transaction was indeed signed by the EOA, the code removes the RLP length prefix and the `r`, `s`, and `v` values with the following code:

```
uint256 totalSignatureSize = RLP_ENCODED_R_S_BYTE_SIZE +
v.encodeUint().length;
uint256 totalPrefixSize = rlpEncodedTx.length - rlpEncodedTxPayloadLen;
bytes memory rlpEncodedTxNoSigAndPrefix =
    rlpEncodedTx.slice(totalPrefixSize, rlpEncodedTx.length -
totalSignatureSize - totalPrefixSize);
```

Notice that this code assumes that `r` and `s` are always encoded as 32-byte values with a 1-byte length prefix, since the hardcoded `RLP_ENCODED_R_S_BYTE_SIZE` variable has a value of 66.

However, this assumption is incorrect. According to the Ethereum yellowpaper, `r`, `s`, and `v` are interpreted as integers when RLP-encoded:

Here, we assume all components are interpreted by the RLP as integer values, with the exception of the access list $T_A$ and the arbitrary length byte arrays $T_i$ and $T_d$.

As specified in the RLP encoding rules, integers are encoded using the shortest possible byte representation, meaning they do not include leading zero bytes.

As a result, if either `r` or `s` has leading zero bytes, its encoded size will be smaller than 33 bytes, which would cause the `totalSignatureSize` to be miscalculated. This would lead to extracting an incorrect transaction payload, which would then be hashed and verified against. Fortunately, the signature would still need to be valid over this incorrect hash, so it would be difficult for an attacker to exploit this issue.

*Note: A real example of this can be seen in the mainnet transaction* `0x0582e3061faa8690c2109fda680c8ffa5eb0f30ed41537b3822cff98a1da4f24`*. Running* `cast tx 0x0582e3061faa8690c2109fda680c8ffa5eb0f30ed41537b3822cff98a1da4f24 --raw` *shows that the* `s` *value is encoded in 32 bytes instead of 33.*

**Recommendation:** To fix this issue, consider passing the `r` and `s` values to the `calculateUnsignedTxHash()` function to explicitly calculate their encoding size using `r.encodeUint().length` and `s.encodeUint().length`, similar to how the `v` value is already handled.

**Biconomy:** Fixed in PR-15.

**Zenith:** Verified.

## [M-2] Potential underflow in `_calculateRefund()`

| Severity: Medium | Status: Resolved |
|---|---|

**Target**

- NodePaymaster.sol#L104-L110

**Severity:**

- Impact: Medium
- Likelihood: Medium

**Description:** In `_calculateRefund()`, NodePaymaster assumes that `maxGasLimit` is greater than `actualGasUsed`, where `actualGasUsed` has been added with a fixed `POST_OP_GAS`.

```
        actualGasUsed = actualGasUsed + POST_OP_GAS;

        // Add penalty
        // We treat maxGasLimit - actualGasUsed as unusedGas and it is
true if preVerificationGas, verificationGasLimit and
pmVerificationGasLimit are tight enough.
        // If they are not tight, we overcharge, as verification part of
maxGasLimit is > verification part of actualGasUsed, but we are ok with
that, at least we do not lose funds.
        // Details:
https://docs.google.com/document/d/1WhJcMx8F6DYkNuoQd75_-
ggdv5TrUflRKt4fMW0LCaE/edit?tab=t.0
        actualGasUsed += (maxGasLimit - actualGasUsed)/10;
```

The `_getMaxGasLimit()` function assumes the maximum gas the `postOp()` call will use is `op.unpackPostOpGasLimit()`, while the `postOp()` call itself assumes it uses `POST_OP_GAS`.

```
    function _getMaxGasLimit(PackedUserOperation calldata op) internal
view returns (uint256) {
        return op.preVerificationGas + op.unpackVerificationGasLimit() +
op.unpackCallGasLimit() + op.unpackPaymasterVerificationGasLimit() +
op.unpackPostOpGasLimit();
    }
```

If `op.unpackPostOpGasLimit()` is less than `POST_OP_GAS`, `actualGasUsed` may be greater than `maxGasLimit`, resulting in underflow.

Note that in the test `op.unpackPostOpGasLimit()` is 45_000, which is less than POST_OP_GAS.

```
        userOp = makeMEEUserOp({
            userOp: userOp,
            pmValidationGasLimit: 22_000,
            pmPostOpGasLimit: 45_000,
            premiumPercentage: 17_00000,
            wallet: userOpSigner,
            sigType: bytes4(0)
        });
```

**Recommendation:** It is recommended to require `op.unpackPostOpGasLimit()` to be greater than `POST_OP_GAS` in `_validatePaymasterUserOp()`.

**Biconomy:** Resolved with PR-14

**Zenith:** Verified.

## 4.2 Low Risk

A total of 2 low risk findings were identified.

### [L-1] Signatures might unintentionally match prefixes

| Severity: Low | Status: Acknowledged |
|---|---|

**Target**

- K1MeeValidator.sol#L149-L161
- K1MeeValidator.sol#L182-L195
- K1MeeValidator.sol#L257-L268

**Severity:**

- Impact: Medium
- Likelihood: Low

**Description:** In the `K1MeeValidator` contract, the `validateUserOp()` and `_validateSignatureForOwner()` functions inspect the first four bytes of the signature to determine which validation library to use. If the first four bytes match a predefined prefix (`SIG_TYPE_SIMPLE`, `SIG_TYPE_ON_CHAIN`, or `SIG_TYPE_ERC20_PERMIT`), the function removes the prefix and passes the remaining signature to the respective validation library. If the first four bytes do not match any prefix, the signature is assumed to belong to the "non-MEE" flow and is used as-is. For example:

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
userOpHash) /* ... */ {
    bytes4 sigType = bytes4(userOp.signature[0:4]);
    // ...
    if (sigType == SIG_TYPE_SIMPLE) {
        return SimpleValidatorLib.validateUserOp(/* ...*/,
userOp.signature[4:], /* ...*/);
    } else if (sigType == SIG_TYPE_ON_CHAIN) {
        return TxValidatorLib.validateUserOp(/* ...*/,
userOp.signature[4:], /* ...*/);
    } else if (sigType == SIG_TYPE_ERC20_PERMIT) {
        return PermitValidatorLib.validateUserOp(/* ...*/,
userOp.signature[4:], /* ...*/);
    } else {
        // fallback flow => non MEE flow => no prefix
        return NoMeeFlowLib.validateUserOp(/* ...*/, userOp.signature, /*
...*/);
```

```
        }
    }
```

Since the "non-MEE" fallback case does not have an expected prefix, there is a chance that the first four bytes of the signature (which would belong to the ECDSA `r` value) could coincidentally match one of the predefined prefixes. If this happens, the signature would be routed to an incorrect validation library and rejected. In this case, the signature would be unusable.

Note that a similar issue exists in `isValidSignatureWithSender()`, which checks the first three bytes of the signature against a three-byte prefix that is shared by the three prefixes mentioned above.

**Recommendation:** Consider adding an explicit prefix for the "non-MEE" case to ensure that its signatures cannot accidentally match one of the predefined prefixes.

**Biconomy:** Acknowledged and commented in PR-17.

**Zenith:** Acknowledged. After a discussion with the team, it was concluded that this scenario is very unlikely (only $3$ out of $2^{32}$ possible 4-byte values would match unintentionally). Since the outcome of this accidental match is only a revert that can be resolved by retrying with a different signature, the code has been left as-is and a comment about this behavior has been added.

## [L-2] Calls to `permit()` may be frontran

| Severity: Low | Status: Resolved |
| --- | --- |

**Target**

- PermitValidatorLib.sol#L104-L108

**Severity:**

- Impact: Low
- Likelihood: Medium

**Description:** In the `PermitValidatorLib`, if `decodedSig.isPermitTx == true`, the library executes the `permit()` call to the ERC20 contract:

```
if (decodedSig.isPermitTx) {
    decodedSig.token.permit(
        expectedSigner, decodedSig.spender, decodedSig.amount,
uint256(decodedSig.superTxHash), vAdjusted, decodedSig.r, decodedSig.s
    );
}
```

It's worth noting that it's possible that an attacker takes the permit signature (e.g. from the public mempool) and calls `permit()` directly on the ERC20 contract before the userOp has a chance to execute. In this case, the `permit()` call in the userOp would fail since the signature would already be used, which would lead to a revert.

Fortunately, this issue does not necessarily invalidate the userOp permanently. Since `isPermitTx` is not committed to in the `userOpHash` or the `superTxHash`, if a frontrun causes a failure when `isPermitTx == true`, the userOp can be resubmitted with `isPermitTx == false`. However despite this workaround, the ability to force a revert if `isPermitTx == true` could still be problematic.

**Recommendation:** Consider whether it is a concern if a userOp can be invalidated by someone frontrunning and using the `permit()` signature directly. If it is a concern, consider doing the `permit()` call in a try/catch block.

**Biconomy:** Fixed in PR-16 by calling `permit()` in a try/catch block.

**Zenith:** Verified.

## 4.3 Informational

A total of 9 informational findings were identified.

### [I-1] Phishing risks and unintended smart account actions

| Severity: Informational | Status: Acknowledged |
|---|---|

**Target**

- TxValidatorLib.sol
- PermitValidatorLib.sol

**Severity:**

- Impact: Medium
- Likelihood: Low

**Description:** In the new Fusion transaction system, users have both their regular EOA and a companion smart contract account. A single signed value can execute an action for the EOA while also authorizing specific userOps to be performed by the smart account.

For example, in the `PermitValidatorLib`, an ERC2612 permit signature from the EOA not only grants a token approval as intended, but it also uses the `deadline` parameter to commit to a merkle root of userOps authorized for the companion smart account. Similarly, in the `TxValidatorLib`, the last 32 bytes of calldata within an RLP-encoded transaction signed by the EOA are interpreted as a merkle root of userOps for the smart account.

With this system of reinterpreting signatures for multiple uses, it's important to consider whether a user might sign a value that could be interpreted as a Fusion transaction for their smart account, even if they did not intend it.

This seems unlikely to happen by accident. The primary data structure for authorizing userOps in the smart account is a merkle tree, and its root depends on the hashing performed by the `MEEUserOpHashLib` and the ERC4337 entrypoint. All of this hashing has a very specific structure, for example the ERC4337 entrypoint includes its own address and the `block.chainid` when computing the `userOpHash`. Since all of these values would ultimately contribute to the merkle root hash, it would be unlikely for another system to coincidentally have the exact same structure.

However, phishing attacks could be a concern. For example, users generally don't pay much attention to the `deadline` parameter when signing permit operations. However in the Fusion system, a maliciously chosen `deadline` could be used to drain the smart account, even if the permit action itself is not malicious. Similarly, the last 32 bytes of calldata from any transaction by the EOA can always be interpreted as a merkle root of userOps. This applies

even if these bytes come from a regular function call rather than being intentionally added as extra data.

**Recommendation:** Consider documenting these potential risks so that users and wallets are aware of them.

**Biconomy:** Acknowledged and documented in PR-17.

### [I-2] Appending extra calldata bytes may not always be possible

| Severity: Informational | Status: Acknowledged |
| --- | --- |

**Target**

- TxValidatorLib.sol

**Severity:**

- Impact: Low
- Likelihood: Low

**Description:** The `TxValidatorLib` is designed for a system where a user sends a standard EOA transaction while appending 32 extra bytes to the end of the calldata. These extra bytes are meant to be a commitment to the root of a merkle tree that contains userOps to be executed in their smart account.

This mechanism assumes that the target contract of the EOA transaction will not revert due to the extra calldata. While most contracts will correctly ignore unexpected trailing data, some may be designed to reject transactions containing additional calldata. Such contracts would not be compatible with the `TxValidatorLib`.

Note that this is a theoretical issue, and no specific examples of incompatible contracts have been identified.

**Recommendation:** This finding has been provided for informational purposes. Since most contracts appear to be compatible with this system, the possibility of a small subset of incompatible contracts does not seem to be a significant concern.

**Biconomy:** Acknowledged and documented in PR-17.

### [I-3] Padding extra data in signatures can increase gas costs

| Severity: Informational | Status: Acknowledged |
| --- | --- |

**Target**

- SimpleValidatorLib.sol#L35-L41

**Severity:**

- Impact: Low
- Likelihood: Medium

**Description:** In most of the verification libraries used by the `K1MeeValidator`, the `signature` bytes can contain extra unused data without affecting the validation logic. For example, in the `SimpleValidatorLib`, the `signature` is decoded as follows:

```
(
    bytes32 superTxHash,
    bytes32[] memory proof,
    uint48 lowerBoundTimestamp,
    uint48 upperBoundTimestamp,
    bytes memory secp256k1Signature
) = abi.decode(signatureData, (bytes32, bytes32[], uint48, uint48,
bytes));
```

Any additional data beyond what is explicitly decoded would be ignored in this logic, meaning it would not impact the validity of the signature. However, including extra data would lead to higher gas costs for passing around the larger signature.

Since the user does not directly submit their userOp on-chain, the transaction sender could theoretically pad a valid signature with extra data to increase the user's gas costs. Fortunately, since the user's maximum gas limits in their userOp must still be respected, this would only allow extracting more value up to the user's predefined limit.

**Recommendation:** Consider whether this behavior is a concern. This is a common issue in ERC4337-related codebases and is hard to fully prevent, as signature sizes vary and detecting unnecessary padding is difficult.

**Biconomy:** Acknowledged.

**Zenith:** Acknowledged. After a discussion with the team, it was decided that this is a minor risk and adding code checks to detect padded signatures is not worth the added complexity.

### [I-4] `TxValidatorLib` does not support type 1 transactions

| Severity: Informational | Status: Acknowledged |
| --- | --- |

**Target**

- TxValidatorLib.sol#L23-L24

**Severity:**

- Impact: Low
- Likelihood: Low

**Description:** The `TxValidatorLib` parses RLP-encoded transactions and currently supports "type 0" (legacy) and "type 2" (EIP-1559) transactions. However, it does not support "type 1" (EIP-2930) transactions.

**Recommendation:** This finding has been provided for informational purposes. Consider documenting this behavior in `TxValidatorLib` by explicitly noting that "type 1" transactions are not supported.

**Biconomy:** Acknowledged and documented in PR-17.

### [I-5] `NodePaymaster` is not compatible with ERC4337 mempool

| Severity: Informational | Status: Acknowledged |
|---|---|

**Target**

- NodePaymaster.sol#L53

**Severity:**

- Impact: Low
- Likelihood: Low

**Description:** In the `NodePaymaster` contract, the `_validatePaymasterUserOp()` function includes a check on `tx.origin`. Since the `ORIGIN` opcode is a banned opcode during the validation stage of an ERC4337 transaction, this makes the `NodePaymaster` incompatible with the general ERC4337 mempool.

After discussing with the team, this behavior was confirmed as intentional, because the `tx.origin` check already restricts the `NodePaymaster` usage to the MEE node owner.

**Recommendation:** Consider documenting this behavior in a comment above the `tx.origin` check. For example:

```
function _validatePaymasterUserOp(PackedUserOperation calldata userOp,
bytes32 userOpHash, uint256 maxCost)
    internal
    virtual
    override
    returns (bytes memory context, uint256 validationData)
{
+   // The use of tx.origin makes the NodePaymaster incompatible with the
general ERC4337 mempool.
+   // This is intentional, and the NodePaymaster is restricted to the
MEE node owner anyway.
    require(tx.origin == owner(), OnlySponsorOwnStuff());
    require(userOp.unpackPostOpGasLimit() >= POST_OP_GAS);
    uint256 premiumPercentage =
uint256(bytes32(userOp.paymasterAndData[PAYMASTER_DATA_OFFSET:]));
    context = abi.encode(userOp.sender, userOp.unpackMaxFeePerGas(),
_getMaxGasLimit(userOp), userOpHash, premiumPercentage);
}
```

**Biconomy:** Acknowledged and documented in PR 17.

### [I-6] `_fillSafeSenders()` input size is not enforced

| Severity: Informational | Status: Resolved |
| --- | --- |

**Target**

- K1MeeValidator.sol#L280-L284

**Severity:**

- Impact: Low
- Likelihood: Low

**Description:** The `_fillSafeSenders()` function takes arbitrary `bytes` input and reads it in 20-byte chunks to populate the `_safeSenders` storage:

```
function _fillSafeSenders(bytes calldata data) private {
    for (uint256 i; i < data.length / 20; i++) {
        _safeSenders.add(msg.sender, address(bytes20(data[20 * i:20 * (i
+ 1)])));
    }
}
```

This function does not have a check to ensure `data.length` is a multiple of 20, so if an invalid length is passed, the last incomplete chunk will be silently ignored. It may be beneficial to add an explicit check for this, in case someone makes a mistake with the `data` they provide.

**Recommendation:** Consider adding a check that `data.length` is a multiple of 20 bytes:

```
function _fillSafeSenders(bytes calldata data) private {
+   require(data.length % 20 == 0);
    for (uint256 i; i < data.length / 20; i++) {
        _safeSenders.add(msg.sender, address(bytes20(data[20 * i:20 * (i
+ 1)])));
    }
}
```

**Biconomy:** Fixed in PR-17 by enforcing the `data` length is a multiple of 20.

**Zenith:** Verified.

### [I-7] `_adjustV()` logic is unnecessary in `PermitValidatorLib`

| Severity: Informational | Status: Resolved |
| --- | --- |

**Target**

- PermitValidatorLib.sol#L173-L185

**Severity:**

- Impact: Low
- Likelihood: Low

**Description:** In the `PermitValidatorLib`, the ECDSA `v` value is taken directly from the signature and transformed using `_adjustV()`:

```
function _adjustV(uint256 v) private pure returns (uint8) {
    if (v >= EIP_155_MIN_V_VALUE) {
        return uint8((v - 2 * _extractChainIdFromV(v) - 35) + 27);
    } else if (v <= 1) {
        return uint8(v + 27);
    } else {
        return uint8(v);
    }
}
```

This transformation is unnecessary because `v` is only used within `PermitValidatorLib` for ECDSA verification. In other libraries (such as the `TxValidatorLib`), the `_adjustV()` function is useful for handling RLP-encoded transactions where `v` can encode extra information. However this is not relevant in the `PermitValidatorLib`, where `v` is simply a parity bit that is directly taken from the signature.

**Recommendation:** Consider removing the `_adjustV()` logic from `PermitValidatorLib` so that `v` is directly passed into the ECDSA verification.

**Biconomy:** Fixed in PR-13.

**Zenith:** Verified.

## [I-8] `onUninstall()` does not clear `_safeSenders` storage

| Severity: Informational | Status: Resolved |
| --- | --- |

**Target**

- K1MeeValidator.sol#L91-L93

**Severity:**

- Impact: Low
- Likelihood: Medium

**Description:** The `K1MeeValidator` contract has two main storage variables: `smartAccountOwners` (which maps each smart account to its associated EOA) and `_safeSenders` (a set of addresses that include the smart account in the hash during ERC1271 verification, which helps prevent signature replay attacks).

However, in the `onUninstall()` function, only the `smartAccountOwners` storage is cleared, and the `_safeSenders` remains unchanged. This means the `_safeSenders` storage persists across uninstallation and reinstallations.

This may be intentional if `_safeSenders` are assumed to remain valid indefinitely.

**Recommendation:** Consider if the `onUninstall()` function should also clear the `_safeSenders` storage, and consider documenting this behavior.

**Biconomy:** Fixed in PR-17 by clearing the account's `_safeSenders` storage in `onUninstall()`.

**Zenith:** Verified. Note that this change may increase the gas cost of `onUninstall()` arbitrarily, depending on the size of the account's `_safeSenders` storage. This can always be mitigated by calling `removeSafeSender()` multiple times before fully uninstalling, so this does not introduce significant risks.

### [I-9] `tryRecover()` considerations

| Severity: Informational | Status: Acknowledged |
| --- | --- |

**Target**

- EcdsaLib.sol#L14-L15

**Severity:**

- Impact: High
- Likelihood: Low

**Description:** The `EcdsaLib` library is used throughout the codebase to verify ECDSA signatures. This library uses the `tryRecover()` function from the Solady library. It is worth noting that `tryRecover()` does not revert on invalid signatures (e.g. when `v` is not 27 or 28), but instead returns `address(0)` when an error happens. So, to avoid false positives, it's important that the result of `tryRecover()` is never compared against `address(0)`.

Currently, it does not seem possible for the `EcdsaLib` library to ever compare against `address(0)`. One reason for this is because the `getOwner()` function in the `K1MeeValidator` contract ensures that the `address(0)` case is specially handled as follows:

```
function getOwner(address smartAccount) public view returns (address) {
    address owner = smartAccountOwners[smartAccount];
    return owner == address(0) ? smartAccount : owner;
}
```

However, since this logic is separate from the `EcdsaLib`, adding explicit checks within the library itself may be useful.

**Recommendation:** Consider documenting this behavior of `tryRecover()`, and consider adding a direct check in `EcdsaLib` to ensure that `address(0)` is never checked against.

**Biconomy:** Acknowledged and documented in PR-17.