Pashov Audit Group

Biconomy Security Review

October 16th 2025 - October 18th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Nexus and Composability

Nexus is Biconomy's ERC-7579–based modular smart account system enabling cross-chain interoperability and pluggable modules for gas abstraction, batching, and session management under ERC-4337 and ERC-7484 compliance. Biconomy Composability lets developers build dynamic, multi-step, type-safe transactions with chaining, validation, and return handling directly from the frontend without on-chain coding.

5. Executive Summary

A time-boxed security review of the **bcnmy/composability**, **bcnmy/nexus** and **bcnmy/mee-contracts** repositories was done by Pashov Audit Group, during which **0xl33**, **0xunforgiven**, **0xTheBlackPanther** engaged to review **Nexus and Composability.** A total of **4** issues were uncovered.

Protocol Summary	
Project Name	Nexus and Composability
Protocol Type	Transaction Builder and Account Abstraction
Timeline	October 16th 2025 - October 18th 2025

Review commit hashes:

- ed0b8d092a49b11aa6a9ed54291c450e16c9fce7 (bcnmy/composability)
- 0b7a7c0f87677ed2abd0d4d3910290c59423c496 (bcnmy/nexus)
- 8054af00e3e44c7d8081c7898e3e9a59843d0ac0 (bcnmy/mee-contracts)

Fixes review commit hashes:

- 5ec5d4759196d83b0a233a0d64a2c74241e08270 (bcnmy/composability)
- 0b64d292db2b5843d7f8f30419e9c481f137628d (bcnmy/nexus)
- ec77b08dc2e975af9b36d78296f55ebc3cbf4129 (bcnmy/mee-contracts)

Scope

`ComposableExecutionBase.sol` `ComposableExecutionLib.sol` `ComposableExecutionModule.sol` `IComposableExecution.sol` `DataTypes.sol` `Constants.sol` `Nexus.sol` `INexusEventsAndErrors.sol` `ModuleManager.sol` `PermitValidatorLib.sol` `SimpleValidatorLib.sol` `TxValidatorLib.sol` `RLPEncoder.sol` `HashLib.sol` `MEEUserOpHashLib.sol` `K1MeeValidator.sol`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	3
Total findings	**4**

Summary of findings

ID	Title	Severity	Status
[M-01]	Non-standard EIP-712 encoding in struct case	Medium	**Resolved**
[L-01]	No signature deadline in `initializeAccount()`	Low	**Resolved**
[L-02]	Missing length check in `BALANCE` fetcher	Low	**Resolved**
[L-03]	Domain separator prevents cross-chain SuperTransaction signatures	Low	**Resolved**

Medium findings

[M-01] Non-standard EIP-712 encoding in struct case

Severity

Impact: Low

Likelihood: High

Description

In `HashLib.compareAndGetFinalHash()`, the struct case uses non-standard EIP-712 encoding that includes ABI encoding prefixes (offset and length), while the array case correctly implements standard EIP-712 encoding.

Comment says: "if SuperTx is a struct, then encoded data is just the concat of all the itemHashes"
Actual code:

```
bytes memory encodedData = abi.encode(itemHashes);  // Adds [offset][length] prefix
structHash = keccak256(abi.encodePacked(outerTypeHash, encodedData));
```

The `abi.encode(itemHashes)` produces: `[offset][length][item0][item1][item2]...`, but per EIP-712 specification (Definition of encodeData), struct field encoding should be direct concatenation (without ABI prefixes): `typeHash ‖ item0 ‖ item1 ‖ item2 ‖ ...`

The mechanics of encoding dynamic arrays in solidity can be found here and here. Tldr; `abi.encode` adds the offset and length prefixes, while `abi.encodePacked` does not.

Impact:

If SDK uses standard EIP-712, signature verification in struct case will always fail (medium-high impact).

If SDK accounts for this inconsistency, signatures verify correctly, but implementation is non-compliant with EIP-712 standard (low impact).

For context: EIP-712 hash creation for SuperTx has not been implemented on the SDK level yet.

Recommendations

Use direct concatenation:

```
} else {
    // if SuperTx is a struct, then encoded data is just the concat of all the itemHashes
-   bytes memory encodedData = abi.encode(itemHashes);
```

```
    /// forge-lint:disable-next-line(asm-keccak256)
-   structHash = keccak256(abi.encodePacked(outerTypeHash, encodedData));
+   structHash = keccak256(abi.encodePacked(outerTypeHash, itemHashes));
}
```

Low findings

[L-01] No signature deadline in `initializeAccount()`

Function `initializeAccount()` allows for a 7702 account to be initialized by relayer if the `initdata` is signed be EOA. The issue is that there's no deadline check for signed messages and it's possible to initiate 7702 multiple times. Suppose users signs initialize transaction and that transaction wasn't executed and users decide to initialize with new parameters/call and create and execute another initialize transaction (with relayer or user operation) then contract would be initialized but the first initialization transaction can be executed any time. Initializing the account with that transaction can result in unexpected result. Either the signed message should have a deadline and checked for execution or it shouldn't be possible to reinitialize account for relayers.

[L-02] Missing length check in `BALANCE` fetcher

The newly introduced `BALANCE` fetcher reads two packed addresses (40 bytes total) from `paramData` using assembly, but never validates the input length before performing the read operations.

if you check `contracts/ComposableExecutionLib.sol` in the `processInput()` function there is no length check of paramData

```
} else if (param.fetcherType == InputParamFetcherType.BALANCE) {
    address tokenAddr;
    address account;
    bytes calldata paramData = param.paramData;
    // expect paramData to be abi.encodePacked(address token, address account)
    assembly {
        tokenAddr := shr(96, calldataload(paramData.offset))           // No length check
        account := shr(96, calldataload(add(paramData.offset, 0x14)))  // Reads unchecked offset
    }
```

The bug manifests when `paramData.length != 40`

- **< 40 bytes**: the second `calldataload` reads beyond the slice boundary into adjacent calldata or zero-padding, producing a garbage/zero address for `account`.
- **> 40 bytes**: then extra bytes are silently ignored without error.
- EVM's `calldataload` does not revert on out-of-bounds reads; it returns zeros or adjacent memory, causing silent failures.

Balance queries execute with incorrect addresses (often `address(0)`) instead of reverting, returning wrong balance values that flow into execution logic undetected, potentially bypassing constraints or causing unintended swaps.

This issue is actually **introduced in** PR #5, commit `[fad744b](https://github.com/bcnmy/composability/pull/5/commits/fad744bce5295c5e2f018e662b60a080ebc8b9a2)` - This is new functionality added to support runtime balance fetching.

Recommendation

Add explicit length validation before the assembly block.

```
} else if (param.fetcherType == InputParamFetcherType.BALANCE) {
    bytes calldata paramData = param.paramData;

    // Validate exact length requirement
    if (paramData.length != 40) {
        revert InvalidParameterEncoding("BALANCE fetcher requires exactly 40 bytes");
    }

    address tokenAddr;
    address account;
    assembly {
        tokenAddr := shr(96, calldataload(paramData.offset))
        account := shr(96, calldataload(add(paramData.offset, 0x14)))
    }
    // ... rest of balance logic
}
```

[L-03] Domain separator prevents cross-chain SuperTransaction signatures

The finding was added after the fix review

The `HashLib.hashTypedDataForAccount()` function constructs the EIP-712 domain separator using `chainId` and `verifyingContract` values retrieved from the account's `eip712Domain()`. This causes the final EIP-712 hash to differ across chains, breaking the core functionality of MEE SuperTransactions which require a single signature to be valid across multiple chains.

```
function hashTypedDataForAccount(address account, bytes32 structHash) internal view returns
(bytes32 digest) {
    (
        /*bytes1 fields*/
        ,
        string memory name,
        string memory version,
        uint256 chainId,              // Chain-specific value
        address verifyingContract,    // May differ per chain
        /*bytes32 salt*/
        ,
        /*uint256[] memory extensions*/
    ) = IERC5267(account).eip712Domain();

    // ...
    mstore(add(m, 0x60), chainId)
    mstore(add(m, 0x80), verifyingContract)
    // ...
}
```

MEE SuperTransactions are designed for users to sign once and have that signature valid across all chains where the operations will execute. The current implementation forces users to sign separately for each chain, defeating the primary UX benefit of the MEE system.

Both `chainId` and `verifyingContract` address are already included in the `userOpHash`, which is computed as part of the MEE validation flow by the Entrypoint contract. This provides replay protection without needing these values in the domain separator.

Modify `hashTypedDataForAccount()` to use fixed values for cross-chain compatibility:

```
function hashTypedDataForAccount(address account, bytes32 structHash) internal view returns
(bytes32 digest) {
    (
        /*bytes1 fields*/
        ,
        string memory name,
        string memory version,
-       uint256 chainId,
-       address verifyingContract,
+       /*uint256 chainId*/
+       ,
+       /*address verifyingContract*/
+       ,
        /*bytes32 salt*/
        ,
        /*uint256[] memory extensions*/
    ) = IERC5267(account).eip712Domain();

    /// @solidity memory-safe-assembly
    assembly {
        //Rebuild domain separator out of 712 domain
        let m := mload(0x40)
        mstore(m, _DOMAIN_TYPEHASH)
        mstore(add(m, 0x20), keccak256(add(name, 0x20), mload(name)))
        mstore(add(m, 0x40), keccak256(add(version, 0x20), mload(version)))
-       mstore(add(m, 0x60), chainId)
-       mstore(add(m, 0x80), verifyingContract)
+       mstore(add(m, 0x60), 0)  // chainId = 0 for cross-chain compatibility
+       mstore(add(m, 0x80), 0)  // verifyingContract = address(0)
        digest := keccak256(m, 0xa0)

        // Hash typed data
        mstore(0x00, 0x1901000000000000)
        mstore(0x1a, digest)
        mstore(0x3a, structHash)
        digest := keccak256(0x18, 0x42)
        // Restore the part of the free memory slot that was overwritten.
        mstore(0x3a, 0)
    }
}
```