Pashov Audit Group

Biconomy Security Review

November 26th 2025 - November 27th 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

Biconomy is a modular infrastructure platform that simplifies how users and developers interact with blockchain applications through account abstraction and interoperability. It enables seamless cross-chain transactions, flexible smart accounts, and dynamic, multi-step interactions powered by composable, type-safe tooling.

5. Executive Summary

A time-boxed security review of the **bcnmy/stx-contracts** repository was done by Pashov Audit Group, during which **unforgiven**, **0xl33**, **ni8mare** engaged to review **Nexus and Composability**. A total of **5** issues were uncovered.

Protocol Summary	
Project Name	Nexus and Composability
Protocol Type	Transaction Builder and Account Abstraction
Timeline	November 26th 2025 - November 27th 2025

Review commit hash:
- b3695fcd76c13fcac27564a2f62ea21252cfe495 (bcnmy/stx-contracts)

Fixes review commit hash:
- 8dea408c0e4ecdcc240a287316e94c5bfb74dac4 (bcnmy/stx-contracts)

Scope

`ComposableExecutionLib.sol` `ComposableStorage.sol` `ComposableExecutionModule.sol` `INexus.sol` `INexusEventsAndErrors.sol` `IBaseAccount.sol` `IExecutionHelper.sol` `IModuleManagerEventsAndErrors.sol` `Nexus.sol` `BaseAccount.sol` `ModuleManager.sol` `K1MeeValidator.sol`

6. Findings

Findings count

Severity	Amount
High	1
Low	4
Total findings	**5**

Summary of findings

ID	Title	Severity	Status
[H-01]	Incorrect assembly packing in `getNamespace` causes collisions	High	Resolved
[L-01]	Assembly errors not declared in the interface	Low	Resolved
[L-02]	`PREPInitialized` event emitted but not declared	Low	Resolved
[L-03]	Free memory pointer not updated	Low	Resolved
[L-04]	Invalid EIP-712 Domain Typehash	Low	Resolved

High findings

[H-01] Incorrect assembly packing in `getNamespace` causes collisions

Severity

Impact: High

Likelihood: Medium

Description

The `getNamespace` function contains incorrect assembly code that can lead to namespace collisions due to improper memory packing.

```
function getNamespace(address account, address _caller) public pure returns (bytes32 result) {
    assembly {
        mstore(0x00, account)
        mstore(0x14, _caller)
        result := keccak256(0x0c, 0x28)
    }
}
```

Memory Layout Analysis:

`mstore(0x00, account)` writes 32 bytes:

- Positions 0x00-0x0b: 12 zero bytes (padding).
- Positions 0x0c-0x1f: 20 bytes of account address.

`mstore(0x14, _caller)` writes 32 bytes:

- Positions 0x14-0x1f: 12 zero bytes (padding) - **This overwrites the last 12 bytes of the account address!**
- Positions 0x20-0x33: 20 bytes of caller address.

`keccak256(0x0c, 0x28)` hashes 40 bytes starting at position 0x0c:

- Positions 0x0c-0x13: First 8 bytes of account address.
- Positions 0x14-0x1f: 12 zero bytes (not part of original account).
- Positions 0x20-0x33: 20 bytes of caller address.

Result: Only 8 bytes of the account address are included in the hash instead of the full 20 bytes. This creates namespace collisions where any two accounts sharing the same first 8 bytes will generate identical namespaces for the same caller.

Additionally, this affects external systems that compute namespaces correctly using `keccak256(abi.encodePacked(account, caller))`. When they call `readStorage()`, the call will revert with `SlotNotInitialized()` because the slot was never initialized with the correctly computed namespace.

Recommendations

Option 1: Use Bit-Shifting in Assembly

```
function getNamespace(address account, address _caller) public pure returns (bytes32 result) {
    assembly {
        mstore(0x00, account)
        mstore(0x20, shl(96, _caller))
        result := keccak256(0x0c, 0x28)
    }
}
```

Option 2: Proper Memory Packing Without Overlap

```
function getNamespace(address account, address _caller) public pure returns (bytes32 result) {
    assembly {
        mstore(0x00, shl(96, account))
        mstore(0x14, shl(96, _caller))
        result := keccak256(0x00, 0x28)
    }
}
```

Both options ensure proper concatenation of the two addresses without padding zeros interfering with the hash computation.

Low findings

[L-01] Assembly errors not declared in the interface

Errors like `EnableModeSigError` and `ValidatorNotInstalled` are being emitted in `assembly`, which encodes the correct error selectors (0x46fdc333 and 0x6859e01e). But these errors are not defined in `IModuleManagerEventsAndErrors` and hence not included in the ABI, and hence this could lead to front-end integration issues.

Recommendation: Include the definitions of these errors or any other errors (like `InvalidPREP`, `CanNotRemoveLastValidator` etc..) that are not defined but are being used in the assembly.

[L-02] `PREPInitialized` event emitted but not declared

In Nexus.sol, take a look at the `installModule` function:

```
        _installModule(moduleTypeId, module, initData);
        assembly {
            // emit ModuleInstalled(moduleTypeId, module)
            mstore(0x00, moduleTypeId)
            mstore(0x20, module)
            log1(0x00, 0x40, 0xd21d0b289f126c4b473ea641963e766833c2f13866e4ff480abd787c100ef123)
        }
```

It emits `ModuleInstalled`, which the protocol has defined in the Constants.sol function.

But, this is not the case for `PREPInitialized(r)`:

```
            // emit PREPInitialized(r)
            mstore(0x00, r)
            log1(0x00, 0x20,
0x4f058962bce244bca6c9be42f256083afc66f1f63a1f9a04e31a3042311af38d) //@audit - missing event
definition?
        }
```

The ABI won't include this event. So, it becomes tough to create event filters, and any user or dapp tracking PREP initialisation will not be able to do so easily.

Recommendation: Declare the `PREPInitialized` event in the `INexusEventsAndErrors` file

[L-03] Free memory pointer not updated

Code builds multiple EIP-712 hashes by borrowing the free-memory pointer, but it doesn't advance it afterward. This is resulting in the remaining of dirty bytes in memory and an out of sync free memory pointe.

```
            // Calculate the hash of the initData
            bytes32 initDataHash;
            assembly {
                let ptr := mload(0x40)
                calldatacopy(ptr, initData.offset, initData.length)
                initDataHash := keccak256(ptr, initData.length)
            }
            // Make sure the account has not been already initialized
```

It happens in multiple functions like `ModuleManager._getEnableModeDataHash()` and `Nexus.initializeAccount()` It is recommended to use a consistent pattern and call `mstore(0x40, add(ptr, size))` after deriving each hash in order to keep the allocator state predictable everywhere. This does not lead to issues by itself, but `solc` relies on `fmp` pointing to free memory, and may lead to bugs/vulns if this is not true. https://docs.soliditylang.org/en/latest/assembly.html#advanced-safe-use-of-memory.

[L-04] Invalid EIP-712 Domain Typehash

The `_DOMAIN_TYPEHASH` is computed from a malformed type string missing a closing parenthesis: `"EIP712Domain(string name"` This produces an incorrect typehash. The **correct** typehash for `"EIP712Domain(string name)"` is: `0xb2178a58fb1eefb359ecfdd57bb19c0bdd0f4e6eed8547f46600e500ed111af3`

Also, as per eip712 if the struct definition involves nested structs, their definition should be appended as well. But, this is not the case currently and results in incorrect hash -

```
// keccak256("SuperTx(MeeUserOp[] meeUserOps)");
bytes32 constant SUPER_TX_MEE_USER_OP_ARRAY_TYPEHASH =
    0x07bdf0267970db0d5b9acc9d9fa8ef0cbb5b543fb897017542bfb306f5e46ad0
```

Hash needs to be calculated as follows -

```
keccak256("SuperTx(MeeUserOp[] meeUserOps)MeeUserOp(bytes32 userOpHash,uint256
lowerBoundTimestamp,uint256 upperBoundTimestamp)");
```

Recommendations

- Replace the incorrect `_DOMAIN_TYPEHASH` with the valid one above.
- Change the comment to `// keccak256("EIP712Domain(string name)";`
- Also, calculate the hash for the nested struct as shown above.