Nexus Security Review

Pashov Audit Group

Conducted by: ast3ros, 0x37, zark, sl1, 0xAbhay, 0xBugSlayer

March 21st 2025 - March 25th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **bcnmy/nexus** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Nexus

Nexus is Biconomy's implementation of ERC-7579 modular smart accounts, providing a standardized framework for account abstraction with native support for cross-chain interoperability. The architecture enables pluggable modules for gas abstraction, transaction batching, and session key management while maintaining compliance with ERC-4337 and ERC-7484 standards.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 6395f8fdd7ff7ddd8b4a1c005c5f9a18d854a2de

fixes review commit hash - 9af2d1d2d71c45b1a519000f51ae39a8944322f3

Scope

The following smart contracts were in scope of the audit:

- `BaseAccount`
- `ExecutionHelper`
- `ModuleManager`
- `RegistryAdapter`
- `Storage`
- `Stakeable`
- `BiconomyMetaFactory`
- `K1ValidatorFactory`
- `NexusAccountFactory`
- `LocalCallDataParserLib`
- `AssociatedArrayLib`
- `BootstrapLib`
- `BytesLib`
- `EnumerableMap4337`
- `EnumerableSet4337`
- `ExecLib`
- `Initializable`
- `ModeLib`
- `ModuleTypeLib`
- `NonceLib`
- `ProxyLib`
- `interfaces/`
- `K1Validator`
- `Constants`
- `DataTypes`
- `Nexus`

7. Executive Summary

Over the course of the security review, ast3ros, 0x37, zark, sl1, 0xAbhay, 0xBugSlayer engaged with Biconomy to review Nexus. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	Nexus
Repository	https://github.com/bcnmy/nexus
Date	March 21st 2025 - March 25th 2025
Protocol Type	Account Abstraction

Findings Count

Severity	**Amount**
Medium	2
Low	13
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Hook bypass via enable mode in module installation	Medium	Resolved
[M-02]	Users can delete all validators from Nexus account	Medium	Resolved
[L-01]	Storage pattern inconsistency exposes registry variable	Low	Resolved
[L-02]	handlePREP() lacks sufficient length validation	Low	Resolved
[L-03]	Nexus checkERC7739Support() checks DEFAULT_VALIDATOR even if unused	Low	Acknowledged
[L-04]	withRegistry modifier missing for validateUserOp() validator	Low	Acknowledged
[L-05]	createAccount() skips isModuleAllowed for prevalidation hooks	Low	Resolved
[L-06]	Cannot opt out of hook in createAccount()	Low	Resolved
[L-07]	RegistryFactory::removeAttester can reduce attester count under limit	Low	Resolved
[L-08]	Outdated emergencyUninstallTimelock may affect emergencyUninstallHook	Low	Acknowledged
[L-09]	Validators may fail to be uninstalled	Low	Acknowledged
[L-10]	Hook check may be bypassed	Low	Resolved
[L-11]	Cannot install preValidationHooks in NexusBoostrap	Low	Resolved
[L-12]	User can bypass the withHook modifier	Low	Resolved
[L-13]	MODULE_ENABLE_MODE_TYPE_HASH is incompatible with eip-712	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Hook bypass via enable mode in module installation

Severity

Impact: High

Likelihood: Low

Description

The Nexus smart account uses a hook mechanism to implement pre-checks and post-checks for operations like module installation. However, there's a vulnerability where these hooks can be bypassed when modules are installed via the Module Enable Mode in `validateUserOp`.

When a module is installed through the normal flow using `installModule`, the `withHook` modifier is applied, which properly calls the hook's `preCheck` and `postCheck` functions:

```
function installModule(
  uint256moduleTypeId,
  addressmodule,
  bytescalldatainitData
) external payable onlyEntryPointOrSelf {
    _installModule(moduleTypeId, module, initData);
    emit ModuleInstalled(moduleTypeId, module);
}

function _installModule(
  uint256moduleTypeId,
  addressmodule,
  bytescalldatainitData
) internal withHook {
    ...
}
```

The hook’s preCheck and postCheck receive:

- msg.sender: The caller (e.g., the user or EntryPoint).
- msg.data: The calldata for installModule, including its function selector.

In this case, the hook can correctly identify that a module installation is happening because msg.data contains the installModule selector. It can then apply the intended security checks.

However, when installing a module via Module Enable Mode in `validateUserOp`, the execution path is different:

`validateUserOp` calls internal function `_enableMode`.

```
function validateUserOp(
        PackedUserOperation calldata op,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    )
        external
        virtual
        payPrefund(missingAccountFunds)
        onlyEntryPoint
        returns (uint256 validationData)
    {
        ...
        } else if (op.nonce.isModuleEnableMode()) {
            // if it is module enable mode, we need to enable the module first
            // and get the cleaned signature
            userOp.signature = _enableMode(userOpHash, op.signature);
        ...
    }
```

The internal function `_enableMode` calls interal function `_installModule`.

```
function _enableMode(
      bytes32userOpHash,
      bytescalldatapackedData
    ) internal returns (bytes calldata userOpSignature
        ...
        _installModule(moduleType, module, moduleInitData);
    }
```

`_installModule` is called and triggers `withHook`.

```
function _installModule(
      uint256moduleTypeId,
      addressmodule,
      bytescalldatainitData
    ) internal withHook {
        ...
    }
```

Although `_installModule` has the `withHook` modifier, the hook's preCheck and postCheck receive the following context:

- msg.sender: EntryPoint contract (not the user's address)
- msg.data: Contains the function selector for `validateUserOp`, not `installModule`

This context mismatch creates a vulnerability: any hook that makes security decisions based on the function selector will fail to identify that a module installation is occurring. The hook might allow the operation to proceed because it sees a `validateUserOp` call instead of an `installModule` call, effectively bypassing the hook's protection.

```
modifier withHook() {
        ...
                tstore(HOOKING_FLAG_TRANSIENT_STORAGE_SLOT, 1)
            }
            bytes memory hookData = IHook(hook).preCheck
              (msg.sender, msg.value, msg.data);
            _;
            IHook(hook).postCheck(hookData);
        }
    }
```

Recommendations

Call to `installModule` external function instead of `_installModule` internal function.

[M-02] Users can delete all validators from `Nexus` account

Severity

Impact: High

Likelihood: Low

Description

In the Nexus smart account system, validators are essential for authorizing transactions through the `validateUserOp` function. The `_uninstallValidator` function allows users to remove validators from their account. If a user removes all validators, the account falls back to a `_DEFAULT_VALIDATOR`. However, if this default validator is not initialized —possible if the user did not configure it during account setup—the account becomes unusable. Specifically, the `validateUserOp` function will fail due to the absence of any active validators, resulting in the user being locked out of their account.

```
function _uninstallValidator
      (address validator, bytes calldata data) internal virtual {
        SentinelListLib.SentinelList storage validators = _getAccountStorage
          ().validators;
        (address prev, bytes memory disableModuleData) = abi.decode(data,
          (address, bytes));


        // Perform the removal first
        validators.pop(prev, validator);

        validator.excessivelySafeCall(gasleft(), 0, 0, abi.encodeWithSelector
          (IModule.onUninstall.selector, disableModuleData));
    }
```

Recommendations

Consider implementing:

- Prevent removal of the last validator OR.
- Initialize the default validator on account creation.

8.2. Low Findings

[L-01] Storage pattern inconsistency exposes `registry` variable

Currently, `registry` is only updated by calling itself using `setRegistry`. However, the `RegistryAdapter` contract stores the `registry` storage variable in `slot 0` without using the ERC-7201 namespaced storage pattern that's used throughout the rest of the codebase. This creates a security vulnerability, as the registry variable could be manipulated through delegate calls or during upgrades.

```
abstract contract RegistryAdapter {
    IERC7484 public registry;
```

It could allow a malicious actor to modify the registry address through carefully crafted delegate calls or future implementations might accidentally overwrite this storage slot.

It's recommended for `RegistryAdapter` contract to use the ERC-7201 namespaced storage pattern consistent with the rest of the codebase.

[L-02] `handlePREP()` lacks sufficient length validation

In the `_handlePREP` function, the code checks if data.length is at least 0x61 (97) bytes. However, this validation is incomplete and fails to properly account for the minimum size of initData.

The data structure should contain:

- `saltAndDelegation`: 32 bytes.
- Pointer to `initData`: 32 bytes.
- Pointer to `cleanedSignature`: 32 bytes.
- Length of `initData`: 32 bytes.
- `initData`: at least 4 bytes (function selector).
- Length of `cleanedSignature`: 32 bytes.
- `cleanedSignature`: at least 65 bytes for ECDSA signatures.

Total is 229 bytes.

```
function _handlePREP(bytes calldata data) internal returns
      (bytes calldata cleanedSignature, bytes calldata initData) {
        bytes32 saltAndDelegation;
        // unpack the data
        assembly {
            if lt(data.length, 0x61) { // @audit 97 bytes
                mstore(0x0, 0xaed59595) // NotInitializable()
                revert(0x1c, 0x04)
            }
        ...
    }
```

It's recommended to update the length check to account for the minimum size requirements of all data components at 229 bytes.

[L-03] Nexus `checkERC7739Support()` checks `DEFAULT_VALIDATOR` even if unused

The `checkERC7739Support` function in the `Nexus` contract checks for support of `ERC-7739` by iterating through installed validators and additionally checking the `_DEFAULT_VALIDATOR`. However, this default validator may not be installed at all if, for example, the smart account was initialized using `NexusBootstrap.initNexus`, which allows specifying only custom validators.

```
function checkERC7739Support(
      bytes32hash,
      bytescalldatasignature
    ) public view virtual returns (bytes4
        bytes4 result;
        unchecked {
            // ...
        }
        result = _get7739Version
        //(_DEFAULT_VALIDATOR, result, hash, signature); // check default validator
        return result == bytes4(0) ? bytes4(0xffffffff) : result;
    }
```

Add a check to verify whether `_DEFAULT_VALIDATOR` is installed for this smart account before calling it within `checkERC7739Support`.

[L-04] `withRegistry` modifier missing for `validateUserOp()` validator

The `validateUserOp` function in the `Nexus` smart contract does not perform a `withRegistry` check on the validator module.

```
function validateUserOp(
        PackedUserOperation calldata op,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    )
        external
        virtual
        payPrefund(missingAccountFunds)
        onlyEntryPoint
        returns (uint256 validationData)
    {
```

This is inconsistent with the behavior for example of executor modules, which are validated via the `withRegistry` modifier in `executeFromExecutor`.

```
function executeFromExecutor(
        ExecutionMode mode,
        bytes calldata executionCalldata
    )
        external
        payable
        onlyExecutorModule
        withHook
        withRegistry(msg.sender, MODULE_TYPE_EXECUTOR)
        returns (bytes[] memory returnData)
    {
```

This discrepancy may allow a validator module who is no more verified from `registry` to be used during user operation validation and, eventually, bypass the registry-based attestation checks.

Consider adding a `withRegistry` check for the validator module inside `validateUserOp` using `_handleValidator` for determing the address of the validator.

[L-05] `createAccount()` skips `isModuleAllowed` for prevalidation hooks

The `createAccount` function in the `RegistryFactory` contract verifies whether each module (validators, executors, hook, and fallbacks) is whitelisted using the `_isModuleAllowed` check. However, it does not perform this check for Pre-Validation Hook modules, which are newly supported module types in the `Nexus` account (under `MODULE_TYPE_PREVALIDATION_HOOK_ERC1271` and `MODULE_TYPE_PREVALIDATION_HOOK_ERC4337`).

Consider including `_isModuleAllowed` checks for `MODULE_TYPE_PREVALIDATION_HOOK_ERC1271` and `MODULE_TYPE_PREVALIDATION_HOOK_ERC4337` in the `createAccount` function to ensure consistent way of checking the validity of all modules with the `registry`.

[L-06] Cannot opt out of hook in `createAccount()`

The `NexusBootstrap::_initNexus` and other related initialization functions (e.g., `_initNexusWithDefaultValidatorAndOtherModules`) allow users to opt out of installing a hook module by setting `hook.module` to `address(0)`:

```
if (hook.module != address(0)) {
    _installHook(hook.module, hook.data);
    emit ModuleInstalled(MODULE_TYPE_HOOK, hook.module);
}
```

This is intentional and enables flexible account setups where a hook is not required.

However, even though the `NexusBootstrap` supports this, the factory function `RegistryFactory::createAccount` does not. It unconditionally calls `_isModuleAllowed` on the hook module:

```
require(_isModuleAllowed(hook.module, MODULE_TYPE_HOOK), ModuleNotWhitelisted
  (hook.module));
```

If `hook.module == address(0)` (meaning the user opted out of the hook), this call will revert due to an unwhitelisted module check on the zero address even though the bootstrap logic would have skipped installing it.

To fix this issue and align `RegistryFactory::createAccount` with the bootstrap logic, consider skipping `_isModuleAllowed` when `hook.module == address(0)`:

```
+ if (hook.module != address(0)) {
+     require(_isModuleAllowed
+ (hook.module, MODULE_TYPE_HOOK), ModuleNotWhitelisted(hook.module));
+ }
```

[L-07] RegistryFactory::removeAttester can reduce attester count under limit

The `RegistryFactory` constructor enforces that the initial `threshold` must be less than or equal to the number of provided `attesters`. However, the `removeAttester` function does not enforce this constraint. As a result, it is possible for the attester list to be reduced below the threshold, which could cause serious problems upon the usage registry with `_checkRegistry`.

Consider adding a check in `removeAttester` to ensure the new length of attesters remains greater than or equal to the current threshold.

```
require(attesters.length - 1 >= threshold, "Cannot go below threshold");
```

[L-08] Outdated `emergencyUninstallTimelock` may affect `emergencyUninstallHook`

In smart account, we will uninstall one hook if the hook does not work or is malicious. We have several ways to uninstall one hook, `emergencyUninstallHook`, `uninstallModule`.

In `emergencyUninstallHook`, we have one timelock for uninstall. We can unistall the validator module after we pass the timelock.

Let's consider below scenario:

1. User triggers `emergencyUninstallHook` to prepare to uninstall this hook. And the `emergencyUninstallTimelock` will be recorded in timestamp X.
2. The executor tries to uninstall this hook via `executeFromExecutor`. Then the hook will be removed directly. But we don't reset the `emergencyUninstallTimelock`. The stale `emergencyUninstallTimelock` will be kept in the contract.
3. The validator is installed again in timestamp X + 1 days + 1.
4. If we want to uninstall this validator, we have to wait at least 3 days.

This scenarios' possibility is quite low considering that these actions should be trusted in most cases.

```
function emergencyUninstallHook(
      EmergencyUninstallcalldatadata,
      bytescalldatasignature
    ) external payable {
        if (hookTimelock == 0) {
            // if the timelock hasnt been initiated, initiate it
            accountStorage.emergencyUninstallTimelock[hook] = block.timestamp;
            emit EmergencyHookUninstallRequest(hook, block.timestamp);
        } else if (block.timestamp >= hookTimelock + 3 * _EMERGENCY_TIMELOCK) {
            accountStorage.emergencyUninstallTimelock[hook] = block.timestamp;
            emit EmergencyHookUninstallRequestReset(hook, block.timestamp);
        } else if (block.timestamp >= hookTimelock + _EMERGENCY_TIMELOCK) {
            accountStorage.emergencyUninstallTimelock[hook] = 0;
            _uninstallHook(hook, hookType, deInitData);
            emit ModuleUninstalled(hookType, hook);
        }
    }
```

Recommendation: Clear the `emergencyUninstallTimelock` when we uninstall the hook via `uninstallModule`.

[L-09] Validators may fail to be uninstalled

In smart account, validator, as one module, can be installed or uninstalled. If there is something wrong in the validator, we have to uninstall this validator.

In the function `uninstallModule`, we will trigger `onUninstall` function to notify the validator. In case of unexpected errors from the validator, we use `excessivelySafeCall`. Even if the `onUninstall` function is reverted, we still want to uninstall the validator we consider the validator is not trusted and malicious.

The problem here is that we transfer all `gasleft()` to the validator. Malicious validators may consume all gas. Although we will leave 1/64 gas, we should notice that `uninstallModule` function has one `withHook` modifier. After `excessivelySafeCall`, we need to execute the `IHook(hook).postCheck(hookData);`. If the postcheck is complicated, then the left gas maybe not enough to finish this check.

```
function uninstallModule(
      uint256moduleTypeId,
      addressmodule,
      bytescalldatadeInitData
    ) external payable onlyEntryPointOrSelf withHook {
        // If this module is not installed, we do not need to uninstall this
        // module.
        require(_isModuleInstalled(
          _isModuleInstalled

        ), ModuleNotInstalled(moduleTypeId, module
        if (moduleTypeId == MODULE_TYPE_VALIDATOR) {
            _uninstallValidator(module, deInitData);
        }
    }
    function _uninstallValidator
      (address validator, bytes calldata data) internal virtual {
        // get current validators.
        SentinelListLib.SentinelList storage validators = _getAccountStorage
          ().validators;

        (address prev, bytes memory disableModuleData) = abi.decode(data,
          (address, bytes));

        validators.pop(prev, validator);
        validator.excessivelySafeCall(gasleft(), 0, 0, abi.encodeWithSelector
          (IModule.onUninstall.selector, disableModuleData));
    }
```

Recommendation: Add one `remaining_gas` parameter to make sure that the `postCheck` function has enough gas to finish.

[L-10] Hook check may be bypassed

In Nexus, there is one hook mechanism. We use this hook mechanism to do pre-checks and post-checks of transactions to ensure conditions and state consistency.

In some key functions, such as `execute`, `executeFromExecutor`, we will add this `withHook` modifier. In `withHook` modifier, we will use this `tload/tsotre` to record whether we've already checked the transaction data.

The problem here is that `tload/tstore`'s lifecycle is the whole transaction. When someone can trigger our key functions twice in one transaction, some transaction data will fail to check data via the hook.

For example, if one executor triggeres `executeFromExecutor` twice in one transaction, the second time's data will fail to check the data.

```
modifier withHook() {
        address hook = _getHook();
        bool hooking;
        assembly {
            hooking := tload(HOOKING_FLAG_TRANSIENT_STORAGE_SLOT)
        }
        // If there is not any hook.
        if (hook == address(0) || hooking) {
            _;
        } else {
            assembly {
                tstore(HOOKING_FLAG_TRANSIENT_STORAGE_SLOT, 1)
            }
            bytes memory hookData = IHook(hook).preCheck
              (msg.sender, msg.value, msg.data);
            _;
            IHook(hook).postCheck(hookData);
        }
    }
```

```
function executeFromExecutor(
        ExecutionMode mode,
        bytes calldata executionCalldata
    )
        external
        payable
        onlyExecutorModule // Only installed executor.
        withHook
        withRegistry
        //(msg.sender, MODULE_TYPE_EXECUTOR) // Check msg.sender is MODULE_TYPE_EXECUTOR.
        returns (bytes[] memory returnData)
    {
        ...
    }
```

Recommendation: tstore the `msg.sender`, `msg.data`. If the `msg.data` changes, we should check the new `msg.sender`, and `msg.data`.

[L-11] Cannot install `preValidationHooks` in `NexusBoostrap`

This may be a missing functionality in `NexusBoostrap`. Every other module can be installed form there and since right after the `delegatecall` to the `NexusBoostrap` a `preValidationHook` hook is user should have a choice of installing such hook. This is where it is used:

```
function validateUserOp(
        PackedUserOperation calldata op,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    ) external virtual payPrefund(missingAccountFunds) onlyEntryPoint returns
      (uint256 validationData) {
        address validator;
        PackedUserOperation memory userOp = op;

        if (op.nonce.isValidateMode()) {
            // do nothing special. This is introduced
            // to quickly identify the most commonly used
            // mode which is validate mode
            // and avoid checking two above conditions
        } else if (op.nonce.isModuleEnableMode()) {
            // if it is module enable mode, we need to enable the module first
            // and get the cleaned signature
            userOp.signature = _enableMode(userOpHash, op.signature);
        } else if (op.nonce.isPrepMode()) {
            // PREP Mode. Authorize prep signature
            // and initialize the account
            // PREP mode is only used for the uninited PREPs
            require(!isInitialized(), AccountAlreadyInitialized());
            bytes calldata initData;
            (userOp.signature, initData) = _handlePREP(op.signature);
 boostrap dellcall @>            _initializeAccount(initData);
        }
        validator = _handleValidator(op.nonce.getValidator());
ERC4337 preValHook @>        (
  userOpHash,
  userOp.signature
) = _withPreValidationHook(userOpHash, userOp, missingAccountFunds
        validationData = IValidator(validator).validateUserOp
          (userOp, userOpHash);
    }
```

Implement such option in the `NexusBoostrap`.

[L-12] User can bypass the `withHook` modifier

He can do so by making installing modules from `NexusBoostrap`. This may come in handy when he wants to bypass the gets the job of validating state changes done or when it gets him money for example. Implement withHook modifier in the `NexusBoostrap` functions as it is `RECOMMENDED` by erc7579 standard as well.

[L-13] `MODULE_ENABLE_MODE_TYPE_HASH` is incompatible with eip-712

As of this moment the `MODULE_ENABLE_MODE_TYPE_HASH` looks like this:

```
// keccak256("ModuleEnableMode
//(address module,uint256 moduleType,bytes32 userOpHash,bytes32 initDataHash)")
bytes32
   constant MODULE_ENABLE_MODE_TYPE_HASH = 0xbe844ccefa05559a48680cb7fe805b2ec58df122784191aed1
```

However, this could lead to reverts in the `ModuleManager::_enableMode` function because of the way `structHash` is computed:

```
function _getEnableModeDataHash(
      addressmodule,
      uint256moduleType,
      bytes32userOpHash,
      bytescalldatainitData
    ) internal view returns (bytes32
@>        return keccak256(abi.encode(
  abi.encode

)
    }
```

As per eip712:

> Definition: The dynamic types are bytes and string. These are like the atomic types for the purpose of type declaration, but their treatment in encoding is different.

The encoding treatment is complied with but the declaration in the `typehash` is not, leading to invalid `structHash` in the `_enableMode` function

Recommendations:

Change the `MODULE_ENABLE_MODE_TYPE_HASH` like this:

```
bytes32 constant MODULE_ENABLE_MODE_TYPE_HASH = keccak256("ModuleEnableMode
  (address module,uint256 moduleType,bytes32 userOpHash,bytes initDataHash)")
```

This will keep `MODULE_ENABLE_MODE_TYPE_HASH` compliant with eip712