Biconomy Nexus Security Review

Auditors

Christoph Michel, Lead Security Researcher

Noah Marconi, Lead Security Researcher

Blockdev, Security Researcher

Devtooligan, Security Researcher

Víctor Martínez, Security Researcher

Chinmay Farkya, Associate Security Researcher

2 Day Extension Assisted by

Noah Marconi, Lead Security Researcher

Report prepared by: Lucas Goiriz

March 4, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Biconomy is the world's most advanced ERC-4337 account abstraction infrastructure platform. Leverage our highly customisable transaction infra to build seamless dApps for every user.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Biconomy Nexus according to the specific commit. Any modifications to the code will require a new security review.

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

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 10 days in total, Binconomy engaged with Spearbit to review the biconomy-nexus protocol. In this period of time a total of **61** issues were found.

Summary

Project Name	Binconomy
Repository	biconomy-nexus
Commit	f6313e61
Type of Project	Infrastructure, Account Abstraction
Audit Timeline	Jul 7th to Jul 17th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	4	4	0
Medium Risk	12	10	2
Low Risk	11	7	4
Gas Optimizations	6	6	0
Informational	28	19	9
Total	**61**	**46**	**15**

The Spearbit team reviewed Biconomy's nexus changes holistically on commit hash 773943fb and determined that the concerning issues were resolved and no new issues were identified.

Note: As part of the fixes of this report, the appropriate check was added for hook and fallback module types but not for validators, which are still vulnerable. For more context see the issue "ERC-7484 registry checks missing on calls to modules".

Additional note: *The Spearbit team has added an* ***addendum*** *to the finding titled "`fallback()` logic prevents hook's `postCheck` from getting executed" on March 4th, 2025.*

5 Findings

5.1 High Risk

5.1.1 `fallback()` logic prevents hook's `postCheck` from getting executed

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: It was decided that all functions be hooked with the `withHook()` modifier to enforce pre and post-checks on the hook for all functions on nexus account.

But the hook post-check do not work correctly on the fallback() because of the current logic.

This is the `withHook()` modifier:

```
modifier withHook() {
    address hook = _getHook();
    if (hook == address(0)) {
        _;
    } else {
        bytes memory hookData = IHook(hook).preCheck(msg.sender, msg.value, msg.data);
        _;
        IHook(hook).postCheck(hookData);
    }
}
```

In case a hook exists, it first does a `hook.preCheck()`, then runs through the execution of the relevant function that was called, followed by a `postCheck()` call to the hook contract.

This is the `fallback()` function for the nexus acocunt:

```
fallback() external payable withHook {
    FallbackHandler storage $fallbackHandler = _getAccountStorage().fallbacks[msg.sig];
    address handler = $fallbackHandler.handler;
    CallType calltype = $fallbackHandler.calltype;
    if (handler != address(0)) {
        if (calltype == CALLTYPE_STATIC) {
            assembly {
                calldatacopy(0, 0, calldatasize())

                // The msg.sender address is shifted to the left by 12 bytes to remove the padding
                // Then the address without padding is stored right after the calldata
                mstore(calldatasize(), shl(96, caller()))

                if iszero(staticcall(gas(), handler, 0, add(calldatasize(), 20), 0, 0)) {
                    returndatacopy(0, 0, returndatasize())
                    revert(0, returndatasize())
                }
                returndatacopy(0, 0, returndatasize())
                return(0, returndatasize())
            }
        }
        if (calltype == CALLTYPE_SINGLE) {
            assembly {
                calldatacopy(0, 0, calldatasize())

                // The msg.sender address is shifted to the left by 12 bytes to remove the padding
                // Then the address without padding is stored right after the calldata
                mstore(calldatasize(), shl(96, caller()))

                if iszero(call(gas(), handler, callvalue(), 0, add(calldatasize(), 20), 0, 0)) {
                    returndatacopy(0, 0, returndatasize())
```

```
                    revert(0, returndatasize())
                }
                returndatacopy(0, 0, returndatasize())
                return(0, returndatasize())
            }
        }
    }
    /// @solidity memory-safe-assembly
    assembly {
        let s := shr(224, calldataload(0))
        // 0x150b7a02: `onERC721Received(address,address,uint256,bytes)`.
        // 0xf23a6e61: `onERC1155Received(address,address,uint256,uint256,bytes)`.
        // 0xbc197c81: `onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)`.
        if or(eq(s, 0x150b7a02), or(eq(s, 0xf23a6e61), eq(s, 0xbc197c81))) {
            mstore(0x20, s) // Store `msg.sig`.
            return(0x3c, 0x20) // Return `msg.sig`.
        }
    }
    revert MissingFallbackHandler(msg.sig);
}
```

The problem is that for all calls inside this fallback logic that succeed, an assembly return statement is used to return the data.

The issue with the assembly return statement is that it ends the whole execution then and there and turns away from the current call context, which means that anything after that `return` is skipped from the call.

This causes the `hook.postCheck()` to be skipped for all fallback handled calls that are successful, and it anyway reverts if the fallback-handled call fails.

This is the flow:

- For all fallback handled calls...
- ⇒ reaches `fallback()` on nexus account...
- ⇒ reaches `withHook()` modifier...
- ⇒ `preCheck()` is called and works...
- ⇒ if external call inside fallback is successful there's an assembly return...
- ⇒ call ends and the `postCheck()` inside `withHook()` modifier will never get called.

This will prevent the hook's postCheck from ever being reached, skipping important logic required by the hook to enforce constraints on the calls to the account.

Recommendation: Instead of using a direct assembly return statement, the returned data from the call needs to be cached into a memory variable, followed by the postCheck call to the hook and then later return the cached data using another assembly block at the end.

This also requires the following changes:

- Instead of using withHook modifier, use internal functions to wrap precheck and postcheck calls and use these inside `fallback()` logic.
- Reorganization of the assembly blocks logic.

Have a look at zerodev's kernel for implementation details on Kernel.sol#L178-L215.

Biconomy: Fixed in PR 183.

Spearbit: Fixed.

Spearbit *(addendum)*: Commits df0d0ed2 and 5cde333a were added to address the present finding. The previous fix of returning a bytes array was noted by the project team to be erronous. The revised correction includes the hook and corrected `onERCXXXReceived()` handling.

Only the contents of these two commits were included in the diff review on March 4th, 2025.

The reviewer for these edits: Noah, Victor & Chinmay.

5.1.2 Enable Mode Signature can be replayed

Severity: High Risk

Context: ModuleManager.sol#L168-L171

Description: During enable mode, two validators are used.

1. `validator`: The module to be installed as any module type that can be defined. It must be a `validator` either already before the user op or after enabling it as a validator in enable mode. This `validator` will be used to validate the final `userOp`.
2. `enableModeSigValidator`: This validator is used in `_checkEnableModeSignature` to check the `_getEnableModeDataHash(validator, initData)` for enabling the first validator.

Note that these two validators are independent of each other and might have different trust assumptions and privileges.

While a user operation has a nonce field (that is used in the `userOpHash` and its signature) and the entrypoint checks and increments this nonce to avoid replaying a user operation, the inner `enableModeSignature` does not have any such replay protection.

The same `module, moduleInitData, enableModeSignature` can be used in a different user operation to install the module a second time, for example, after the user uninstalled the module already.

As the entire enable mode data is encoded in the `userOp.signature` that is not part of `userOpHash`, a bundler can replace the enable mode data with a different previously signed one without invalidating the user operation (as long as the enable mode bit and the `validator` encoded in the `userOp.nonce` match).

```
op.signature = (moduleType, moduleInitData, enableModeValidator, enableModeSignature, userOpSignature)
```

> This signature can also be replayed across another chain, as well as on another smart account on the same chain with the same owner.

Recommendation: The enable mode signature validator should implement its own replay protection.

A gas-efficient way is to tie the enable mode signature to a specific user operation (as one enable mode should map to one user operation). This can be done by also signing over the `userOpHash` for the enable mode signature off-chain. On-chain, when `validateUserOp(op, userOpHash, missingAccountFunds)` is called, the passed `userOpHash` is used to reconstruct the message digest:

```
function _getEnableModeDataHash(address module, uint256 moduleType, bytes32 userOpHash, bytes calldata
↪  initData) internal view returns (bytes32 digest) {
    // userOpHash is from validateUserOp
    digest = _hashTypedData(
        keccak256(
            abi.encode(
                // IMPORTANT! NEED TO CHANGE MODULE_ENABLE_MODE_TYPE_HASH TO INCLUDE THE NEW FIELDS
                MODULE_ENABLE_MODE_TYPE_HASH,
                module,
                moduleType,
                userOpHash,
                keccak256(initData)
            )
        )
    );
}
```

As there is replay protection for a `userOpHash` when coming from the entrypoint, there's also replay protection for the enable mode signature when verifying it in `validateUserOp` from the entrypoint.

Biconomy: Fixed in PR 112 and documented in the wiki.

Spearbit: Fixed.

5.1.3 Enable Mode Signature ignores module type

Severity: High Risk

Context: ModuleManager.sol#L168-L171

Description: During enable mode, two validators are used.

1. `validator`: The module to be installed as any module type that can be defined. It must be a `validator` either already before the user op or after enabling it as a validator in enable mode. This `validator` will be used to validate the final `userOp`.
2. `enableModeSigValidator`: This validator is used in `_checkEnableModeSignature` to check the `_getEnableModeDataHash(validator, initData)` for enabling the first validator.

Note that these two validators are independent of each other and might have different trust assumptions and privileges.

The `enableModeSigValidator` owner might only want to produce a signature for a specific module type. However, the signature is only over the `module` (`validator`) and `initData` fields, not over the `moduleType` that the module will be installed as. The user op submitter can forge the module type without invalidating the enable mode signature. (As the entire enable mode data is encoded in the `userOp.signature` that is not part of `userOpHash`).

They can then create a user op with the forged enable mode module type to install the `module` as any type they want.

A multi-type validator can be installed as all of its types.

Recommendation: Consider signing the module type as part of the `_getEnableModeDataHash`.

Biconomy: Fixed PR 112.

Spearbit: Fixed.

5.1.4 ERC-7484 registry checks missing on calls to modules

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The ERC-7484 registry is not checked when calls to validators, fallback handlers, or hooks are made.

The check is performed when executors are used through the use of the `withRegistry` modifier.

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

The check is also performed when any new module is installed.

```
function _installValidator(address validator, bytes calldata data) internal virtual
↪  withRegistry(validator, MODULE_TYPE_VALIDATOR) {
    // ...

function _installExecutor(address executor, bytes calldata data) internal virtual
↪  withRegistry(executor, MODULE_TYPE_EXECUTOR) {
    // ...

function _installHook(address hook, bytes calldata data) internal virtual withRegistry(hook,
↪  MODULE_TYPE_HOOK) {
    // ...

function _installFallbackHandler(address handler, bytes calldata params) internal virtual
↪  withRegistry(handler, MODULE_TYPE_FALLBACK) {
    // ...
```

The smart account owner has the option to opt-in or opt-out of using the ERC-7484 registry by setting the `registry` variable.

```
modifier withRegistry(address module, uint256 moduleType) {
    _checkRegistry(module, moduleType);
    _;
}

function _checkRegistry(address module, uint256 moduleType) internal view {
    IERC7484 moduleRegistry = registry;
    if (address(moduleRegistry) != address(0)) {
        // this will revert if attestations / threshold are not met
        moduleRegistry.check(module, moduleType);
    }
}
```

The `withRegistry` modifier first checks storage to see if an address has been set for `registry`. If it is set, then `check()` is called on the ERC-7484 registry, passing the module and the module type. If the number of attesters does not meet the threshold for this module / type combination, then the transaction reverts.

The ERC-7579 standard states:

> the Adapter SHOULD implement the following functionality: Revert the transaction flow when the Registry reverts. Query the Registry about module A on installation of A. Query the Registry about module A on execution of A.

While the current logic does perform the query on installation, it fails to do so during execution of hooks, fallback handlers, and validations.

If a user opts-in to using the registry, installs a validator that has sufficient support from the ERC7484 attesters, and then subsequently that validator is determined to be unsafe and the attesters remove their attestation, then this unsafe validator will continue to be used by the smart account since there is no check upon usage.

The impact from using such an unsafe validator could be very high. It is reasonable to imagine a scenario where a module was initially deemed safe and later found to be unsafe, causing attesters to change their position. As such the likelihood is deemed to be medium with an overall severity of rating high.

Recommendation: Use the `withRegistry` modifier or otherwise call `check` on the registry for all functions that will invoke calls to hooks, fallback handlers, and validators.

So, according to ERC-7484 the module should be checked against the registry any time is called. This may be through the use of the `withRegistry` modifier or by calling `_checkRegistry` within a function before a module is called.

- `fallback()`: call `_checkRegistry(handler)` before calling the handler.
- `withHook()`: call `_checkRegistry(hook)` before calling the `preCheck` on the hook.

- `validateUserOp()`: call `_checkRegistry(validator)` before calling the `validateUserOp` on the validator.

Biconomy: Acknowledged. I dont think we can do:

> `validateUserOp()`: - call `_checkRegistry(validator)` before calling the `validateUserOp` on the validator.

Addressed in PR 151.

Spearbit: The check has been added to the Hook and Fallback but not Validators as mentioned.

5.2 Medium Risk

5.2.1 Incorrect check in `_enableMode` prevents installing module types other than validator

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: It was decided based on a previous review that the intention is to allow any module types to be installed in Enable Mode.

Current code has a check that prevents any module type other than a validator(or a multi type) from being installed on a nexus account via the enable mode.

```
function _enableMode(bytes32 userOpHash, bytes calldata packedData) internal returns (bytes calldata
↪  userOpSignature) {
    address module;
    uint256 moduleType;
    bytes calldata moduleInitData;
    bytes calldata enableModeSignature;

    (module, moduleType, moduleInitData, enableModeSignature, userOpSignature) =
    ↪  packedData.parseEnableModeData();

    if (!_checkEnableModeSignature(_getEnableModeDataHash(module, moduleType, userOpHash,
    ↪  moduleInitData), enableModeSignature))
        revert EnableModeSigError();

    // Ensure the module type is VALIDATOR or MULTI
        if (moduleType != MODULE_TYPE_VALIDATOR && moduleType != MODULE_TYPE_MULTI) revert
        ↪  InvalidModuleTypeId(moduleType); // <<<

    _installModule(moduleType, module, moduleInitData);
}
```

Recommendation: Remove this check `if (moduleType != MODULE_TYPE_VALIDATOR && moduleType != MODULE_TYPE_MULTI) revert InvalidModuleTypeId(moduleType);`.

Biconomy: Fixed in PR 177.

Spearbit: Fixed.

5.2.2 Timelock on `emergencyUninstallHook()` can be bypassed

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `emergencyUninstallHook()` function is meant to allow the account owner to remove a hook from the account (minus the pre and post checks on the hook).

It is an alternate to the `uninstallModule()` function which offers hook checks on calls to uninstall any type of modules.

A timelock has been placed to only allow the account to emergency-uninstall the hook after 1 day of placing an uninstall request. But `emergencyUninstallHook()` fails to check that the hook was actually installed.

```
    function emergencyUninstallHook(address hook, bytes calldata deInitData) external payable
    ↪  onlyEntryPoint {
    AccountStorage storage accountStorage = _getAccountStorage();
    uint256 hookTimelock = accountStorage.emergencyUninstallTimelock[hook];

    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) {
        // if the timelock has been left for too long, reset it
        accountStorage.emergencyUninstallTimelock[hook] = block.timestamp;
        emit EmergencyHookUninstallRequest(hook, block.timestamp);
    } else if (block.timestamp >= hookTimelock + _EMERGENCY_TIMELOCK) {
        // if the timelock expired, clear it and uninstall the hook
        accountStorage.emergencyUninstallTimelock[hook] = 0;
        _uninstallHook(hook, deInitData);
        emit ModuleUninstalled(MODULE_TYPE_HOOK, hook);
    } else {
        // if the timelock is initiated but not expired, revert
        revert EmergencyTimeLockNotExpired();
    }
}
```

This allows the timelock to be bypassed through the following steps :

- Call `emergencyUninstallHook()` to place an uninstall request even before installing the hook ⇒ this records a timestamp corresponding to the hook address.
- After nearly a day has passed, install the hook and use it.
- Immediately call `emergencyUninstallHook()` to utilize the request that was placed before.
- As only a day has passed, the call will go through.
- The timelock has been effectively bypassed by the account.

Recommendation: `UninstallModule()` on `Nexus.sol` checks that the module they are trying to uninstall through the call is actually installed. Add the same check to `emergencyUninstallHook()`.

Biconomy: Fixed in PR 175.

Spearbit: Fixed.

5.2.3 Not all calls forward `msg.value`

Severity: Medium Risk

Context: ModuleManager.sol#L102

Description: Most functions of the codebase are `payable`. When these functions perform an inner call, the `msg.value` is not always forwarded:

- `Nexus.executeUserOp` does not forward it in `target.call(data)`.
- `Nexus.fallback` handler.
- `BiconomyMetaFactory.deployWithFactory` does not forward it in `factory.call(factoryData)`.
- Other factories don't forward it in `INexus(account).initializeAccount(initData)`.

Native tokens can be stuck in these contracts, in case of the factory they are unrecoverable. The `fallback` handler calls will not work as expected when receiving `msg.value`.

Recommendation: Consider forwarding `msg.value` whenever expected. We believe the `fallback` should forward native tokens to the fallback handler.

The factories initializing the smart account should not require a `msg.value`. Neither does the `executeUserOp` as it's called from the entrypoint with a value of zero.

Biconomy: Fixed in PR 113.

Spearbit: Fixed. The `msg.value` is now forwarded, the factories send it to the deployment and `initNexus` is called with a `msg.value` of 0.

5.2.4 Module can prevent itself from uninstalling

Severity: Medium Risk

Context: ModuleManager.sol#L226, ModuleManager.sol#L244, ModuleManager.sol#L263, ModuleManager.sol#L312, Nexus.sol#L208-L214

Description: When `Nexus.uninstallModule(..., module, ...)` is called to uninstall `module`, `module.unUninstall(...)` is called eventually. `Nexus` can be stopped from uninstalling if `module.onUninstall()` reverts as it reverts the call to `Nexus` as well. This can be an issue if a module is malicious (or is later discovered to be malicious) since they have the privilege of validating or executing a User Operation.

Recommendation: Ignore reverts when `.onUninstall()` is called on a module, and continue the execution of code in `Nexus` contract.

Biconomy: Fixed in PR 117 and PR 143.

Spearbit: Fixed.

5.2.5 Setting the ERC-7484 registry may put modules in a insecure state

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When the settings for the ERC-7484 registry are set, insecure modules may be left inactive or executor modules may become suddenly deactivated.

The `setRegistry` function is used to configure the registry. Configurable options include updating the set of attester addresses, changing the threshold, updating the address of the registry or disabling it altogether by setting the registry to address zero.

```
function _configureRegistry(IERC7484 newRegistry, address[] calldata attesters, uint8 threshold)
↪   internal {
    registry = newRegistry;
    if (address(newRegistry) != address(0)) {
        newRegistry.trustAttesters(threshold, attesters);
    }
    emit ERC7484RegistryConfigured(newRegistry);
}
```

Any one of these options may affect currently installed modules. For example, an executor may not meet the new threshold that was set and would instantly be disabled. Or worse, a validator may not have the required number of attestations and would continue to be active.

Recommendation: When updating the configuration settings, if the registry address is not set to zero then iterate through each module and perform the registry check.

```
  function _configureRegistry(IERC7484 newRegistry, address[] calldata attesters, uint8 threshold)
  ↪  internal {
      registry = newRegistry;
      registry = newRegistry;
      if (address(newRegistry) != address(0)) {
          newRegistry.trustAttesters(threshold, attesters);
+         AccountStorage storage ams = _getAccountStorage();
+
+         // (iterate over validators calling _checkRegistry() for each
+         // (repeat for executors)
+         // (repeat for fallback handlers)
+
+         _checkRegistry(ams.hook, 4);
      }
-     emit ERC7484RegistryConfigured(newRegistry);
  }
```

Note this would also involve tracking the list of fallback handlers.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.2.6 `fallback()` can be used for direct unauthorised calls to fallback handlers

Severity: Medium Risk

Context: ModuleManager.sol#L72

Description: The `fallback()` function in Nexus smart account is meant for handling calls that need to be routed to a fallback handler module corresponding to the selector in the calldata.

It checks if a handler has been installed for the selector extracted from the calldata, and if a handler exists in the account, then makes a call to the registered handler with the supplied calldata.

But the problem is that the `fallback()` function is unrestricted so it can also be called by anyone directly. If an attacker calls the smart account with a `msg.sig == a` particular selector, the call will be routed to the installed fallback handler from the account.

This alternate call path does not check if the entrypoint is the caller, and also does not have any signature validation because there is no `userOp` and no validator modules are involved.

This can compromise the security of the account in many ways. One example could be a handler having logic to use account's funds in some way so the account might have given large approvals of funds to the handler. But now the attacker gets to misuse the approvals by using the calldata he wishes.

Recommendation: All external functions in `Nexus.sol` are restricted to only entrypoint or self except the fallback itself. While the fallback handlers installed by the user themselves might not be malicious, the unrestricted access to `fallback()` can be used by an attacker.

Implement appropriate authorization control in `fallback()` if required, for example, `onlyEntrypoint()` or a set of allowed callers specific to the handler used.

According to Eip-7579, "*fallback function must implement authorization control*" but note that complying with this will restrict the usefulness of fallback handlers. Alternatively, ensure authorization control is implemented in the fallback handler itself, based on the appended `msg.sender`.

Biconomy: Acknowledged.

> Ensure authorization control is implemented in the fallback handler itself, based on the appended `msg.sender`.

We believe this is enough.

I have opened PR 590 this PR on the ERC having discussed with the authors. Once merged here, we will ask for merge in the ethereum/repo.

We're also hooking the fallback function now.

Spearbit: Acknowledged.

5.2.7 Modules other than validators cannot be installed in enable mode

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: According to the documentation, "Any module type can be installed via Module Enable Mode".

This mode is meant for installing a module from within `validateUserOp()` before it gets used in the following execution defined by the `userOp`.

This is helpful in cases where the account owner does not want to issue an explicit `installModule` in the `userOp` `callData` and for use cases where a module installation during the validation might be necessary for the execution to work, as highlighted in the documentation. This is how modules are installed in enable mode:

```
function validateUserOp(
    PackedUserOperation calldata op,
    bytes32 userOpHash,
    uint256 missingAccountFunds
) external virtual payPrefund(missingAccountFunds) onlyEntryPoint returns (uint256 validationData) {
    address validator = op.nonce.getValidator();
    if (!op.nonce.isModuleEnableMode()) {
        // Check if validator is not enabled. If not, return VALIDATION_FAILED.
        if (!_isValidatorInstalled(validator)) return VALIDATION_FAILED;
        validationData = IValidator(validator).validateUserOp(op, userOpHash);
    } else {
        PackedUserOperation memory userOp = op;
        userOp.signature = _enableMode(validator, op.signature);
        validationData = IValidator(validator).validateUserOp(userOp, userOpHash);
    }
```

Contrary to the documentation, this logic will only allow validator modules and multi-type modules with a validator component to be installed in enable mode.

This is because the module address that is to be installed is extracted from nonce, and then the same module address is used for the validation of `userOp`.

This is incorrect because other module types (hooks, executors, and fallback handlers) cannot be expected to validate the userOp. Any standard implementation of these modules would not have a `validateUserOp()` function.

Biconomy intends to allow any module type to be installed in the "Module Enable Mode," but the current logic results in broken functionality.

Recommendation: Currently, the module-to-be-installed will always be used as a validator. Enable mode makes most sense for validators as other module types can be installed via batching as the documentation acknowledges:

> If the module we want to install before the usage is executor, this can be simply solved by batching `installModule(executor)` call before the call that goes through this executor. However, that would not work for validators, as validation phase in 4337 goes before execution, thus the new validator should be enabled before it is used for the validation. Enable Mode allows to enable the module in the beginning of the validation phase. To achieve this, user signs the data object that describes which module is going to be installed and how it should be configured during the installation. Any module type can be installed via Module Enable Mode, however we believe it makes most sense for validators and hooks.

Consider restricting enable mode to validator module type only, otherwise, the current approach of always using the module as a validator needs to be changed significantly.

Biconomy: Fixed in PR 112.

Spearbit: Fixed, there are 3 modules now:

- The outer `validator` that validates the user op, coming from the `userOp.nonce`.
- The `enableModeValidator` and the `module` to be installed, both coming from `userOp.signature`.

If we want to install `moduleType = validator`, we can set `module = validator` and use the new module to validate the `userOp`. (That's the main usecase for enable mode.) If we want to install a non-validator module, these three validators can all be different.

5.2.8 Missing call type validation in `_installFallbackHandler`

Severity: Medium Risk

Context: ModuleManager.sol#L281

Description: When installing a fallback handler, there is no check that the call type is either `CALLTYPE_STATIC` or `CALLTYPE_SINGLE`. This will result in a successful installation including completion of the initialization steps.

However, when the fallback is called nothing will happen when the selector for the fallback handler is passed. Additionally, the fallback does not revert in this case.

This could have a serious effect on an integrator depending on the fallback and it may not be caught immediately due to lack of reverting.

Recommendation: Add a check in `_installFallbackHandler`:

```
require(calltype == CALLTYPE_SINGLE || calltype == CALLTYPE_STATIC), "Invalid CallType");
```

Also consider updating `fallback()` to revert if neither call type is called.

```
          returndatacopy(0, 0, returndatasize())
          return(0, returndatasize())
      }
- }
- if (calltype == CALLTYPE_SINGLE) {
+ } else if (calltype == CALLTYPE_SINGLE) {
      assembly {
          calldatacopy(0, 0, calldatasize())

@@ -106,6 +105,8 @@ abstract contract ModuleManager is Storage, Receiver, EIP712, IModuleManagerEven
          returndatacopy(0, 0, returndatasize())
          return(0, returndatasize())
      }
+ } else { revert UnsupportedCallType(); }
```

Biconomy: Fixed. See files ModuleManager.sol#L305 and ModuleManager.sol#L129.

Spearbit: Fixed.

5.2.9 Cannot install fallback handler for empty calldata

Severity: Medium Risk

Context: ModuleManager.sol#L73

Description: When a call to a contract without any calldata is performed, the `receive` function is executed (if defined), otherwise, the `fallback` function is executed (if defined).

The `Nexus` contract inherits from `Receiver` which defines the `receive` function.

Meaning the contract's `fallback` function will never be called (for empty calldata) and all `fallback` handlers for `msg.sig = bytes4(0)` will be ignored. It's up to interpretation if a fallback handler for `bytes4(0)` should handle empty calldata, it is not specified in EIP-7579.

The contract code indicates that empty calldata should be handled by a fallback handler; the `_isModuleInstalled` function defaults an empty selector to 'bytes4(0):

```
if (additionalContext.length >= 4) {
    selector = bytes4(additionalContext[0:4]);
} else {
    selector = bytes4(0x00000000);
}
```

Furthermore, if no `receive()` function is defined but a `fallback()`, the `msg.sig` of the `fallback` will also indicate `bytes4(0)`.

Recommendation: Clarify the expected behavior regarding fallback handlers and empty calldata. Consider removing the `receive()` function and adding the following logic to `fallback`:

- If `msg.data.length == 0`:
  - If a fallback handler is installed for `bytes4(0)` call that fallback handler.
  - Otherwise, `return` with empty data. By default, this will simulate a `receive()` function that accepts native tokens.

Biconomy: Fixed in PR 132.

Spearbit: Fixed. `bytes(0)` has been disabled when installing fallback handlers.

5.2.10 Cannot install fallback handler for NFT token callbacks

Severity: Medium Risk

Context: ModuleManager.sol#L72

Description: Note that the `receiverFallback` modifier performs a call context `return` when the `msg.sig` equals one of `onERC721Received(address,address,uint256,bytes)`, `onERC1155Received(address,address,uint256,uint256,bytes)`, `onERC1155BatchReceived(address,address,uint256[],ui`
Therefore, these function types cannot use custom fallback handlers.

While fallback handlers for these function types can be installed without errors, they will never be executed. Smart wallets cannot selectively reject NFTs.

Recommendation: Consider checking for fallback handlers first, only if there's no fallback handler installed, check if it matches one of the NFT receiver functions.

Biconomy: Fixed in PR 132.

Spearbit: Fixed.

5.2.11 The `ModuleTypeLib.bitEncode*` functions error on duplicates

Severity: Medium Risk

Context: ModuleTypeLib.sol#L28

Description: When encoding several `ModuleTypes` into a bitmask using the `bitEncode*` functions, the resulting bitmask will be wrong when the module types are not all unique.

For example, encoding the `MODULE_TYPE_VALIDATOR = 1` twice, will result in an encoding of `MODULE_TYPE_EXECUTOR = 2`. This `EncodedModuleTypes` type can be used in modules, like in mock executor's `getModuleTypes() returns (EncodedModuleTypes)`.

Proof of concept:

```
function test_bitEncode_error() public {
    ModuleType[] memory moduleTypes = new ModuleType[](2);
    moduleTypes[0] = ModuleType.wrap(MODULE_TYPE_VALIDATOR);
    moduleTypes[1] = ModuleType.wrap(MODULE_TYPE_VALIDATOR);
    EncodedModuleTypes enc = bitEncode(moduleTypes);
    // these assertions fail
    assertEq(isType(enc, ModuleType.wrap(MODULE_TYPE_VALIDATOR)), true, "should be a validator");
    assertEq(isType(enc, ModuleType.wrap(MODULE_TYPE_EXECUTOR)), false, "should not be an executor");
}
```

Recommendation: Consider rewriting the `ModuleTypeLib` functions to use bitwise `OR` instead of addition.

```
// example for bitEncode, similar adjustments should be done for isType, bitEncodeCalldata
function bitEncode(ModuleType[] memory moduleTypes) internal pure returns (EncodedModuleTypes) {
    uint256 result;

    // Iterate through the moduleTypes array and set the corresponding bits in the result
    for (uint256 i; i < moduleTypes.length; i++) {
        result |= uint256(1) << ModuleType.unwrap(moduleTypes[i]);
    }

    return EncodedModuleTypes.wrap(result);
}
```

Biconomy: Fixed in PR 133.

Spearbit: Fixed.

5.2.12 `RegistryFactory.addAttester` allows duplicate and unsorted attesters breaking ERC-7484 compliance

Severity: Medium Risk

Context: RegistryFactory.sol#L58

Description: The `RegistryFactory.addAttester` function allows adding duplicates and the new attester is pushed as the last element to the internal list. This leads to several issues:

1. Duplicate attesters are not removed by a single `removeAttester` call.
2. Upon deployment of a smart account, when `REGISTRY.check(module, moduleType, attesters, threshold)` is called, duplicates or unsorted elements in `attesters` will revert according to EIP-7484 registry.

   The attesters provided MUST be unique and sorted and the Registry MUST revert if they are not.

Factory deployments can fail if the `attesters` are misconfigured. This misconfiguration will only be found out later, when an actual deployment is performed.

Recommendation: Consider checking that the `attesters` array is unique and sorted. For example, replace `addAttesters` and `removeAttesters` with a `setAttesters(address[] calldata attesters)` function that iterates over the new attesters to ensure these properties.

Biconomy: Fixed in PR 134.

Spearbit: Fixed. The `addAttesters` and `removeAttesters` functions were kept and perform a sort on the updated array now.

5.3 Low Risk

5.3.1 `_checkEnableModeSignature` conditional branches have duplicate bodies

Severity: Low Risk

Context: ModuleManager.sol#L394-L404

Description/Recommendation: The `if/else` branching executes both blocks with identical code regardless of whether the conditional evaluates to true or false.

Biconomy: Fixed in PR 177.

Speabit: Fixed.

5.3.2 `withHook` modifier is used on both the external and internal `installModule` function

Severity: Low Risk

Context: ModuleManager.sol#L186-L187, Nexus.sol#L152

Description: The hook is called multiple times when the external `installModule` function is called.

Hooks not able to be called 2x will revert and may DOS module installation.

Recommendation: The internal function requires the modifier as `_installModule` is called from more than one places.

Remove from the external function with a natspec comment explaining that the modifier is applied to the internal function. Include a second natspec comment on the internal function to indicate that it should not be removed.

Biconomy: Fixed in PR 178.

Spearbit: Fixed.

5.3.3 Refactor nested EIP-712 code

Severity: Low Risk

Context: Nexus.sol#L439

Description: The nested EIP-712 flow inlined functions that can be found in the Solady dependency. Solady's code has diverged by now. For ease of maintenance and to reduce mistakes from copying over the code, consider removing the inlined function and directly calling solady's functions.

Recommendation: Consider refactoring this code by using _erc1271UnwrapSignature followed by _erc1271IsValidSignatureViaNestedEIP712.

Biconomy: Fixed in PR 158.

Speabit: Acknowledged.

5.3.4 `Bootstrap` installs modules before setting registry

Severity: Low Risk

Context: RegistryBootstrap.sol#L46

Description: The `Bootstrap.init*` functions install modules before setting the registry. This will skip the registry check on these modules. Modules that are not validated could be installed during initialization.

Recommendation: Set the registry before installing modules.

Biconomy: Fixed in PR 115.

Spearbit: Fixed.

5.3.5 Missing EIP-165 `supportsInterface` not EIP-7579 compliant

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: EIP-7579 states:

> ERC-165: Smart accounts MUST implement ERC-165. However, for every interface function that reverts instead of implementing the functionality, the smart account MUST return false for the corresponding interface id.

Recommendation: Consider implementing EIP-165's `supportsInterface` as described in EIP-7579 to be compliant.

Biconomy: Acknowledged. We're planning to do this via fallback handlers. Something like this in rhinestonewtf/core-modules.

Spearbit: Acknowledged.

5.3.6 Fallback handler can call into sensitive functions of fallback handler itself

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Nexus smart account allows to install fallback handlers to route calls to certain module contracts based on function selectors in supplied calldata.

When such a handler is installed via `_installModule`, two selectors (for `onInstall` and `onUninstall`) are restricted from being used as fallback selectors to protect the account from being compromised if the fallback handler gets manipulated by the calldata.

But only restricting these two selectors is not enough. Any sensitive selector might need to be prevented from being present in fallback handlers.

For example, if a fallback handler were created for `transferOwnership()` (or anything with that function signature) then anyone could call this function on the module and take ownership. From the module's perspective, it is unknown if this call originated from the SA or from an outside caller via the fallback.

Other `IFallback` functions like `isModuleType` / `isInitialized` are not protected against either, the smart account could wrongly appear as a module itself.

This is similar to a common vulnerability in early cross-chain bridge implementations that allowed calls to arbitrary functions on arbitrary addresses. A common solution for bridges is to restrict the call to a predetermined function selector such as `onReceive()`and leave it to the integrating developer to handle the message routing.

Recommendation: Consider disallowing setting up selectors for any `IFallback` function. EIP-7579 specifies that the fallback "*MUST route to fallback handlers based on the function selector of the calldata*".

If one is willing to be non-compliant, routing any selector to an `IFallback.onCallback(bytes calldata data)` that is invoked with the `msg.data` (and appended `msg.sender`) separates the fallback handler's handler function from its other management functions like `transferOwnership`, reducing the room for error.

Biconomy: Acknowledged. Everything of this is solved by PR 590 to 7579 stating that Fallback modules should implement proper access control on their methods.

This will even allow to remove the line restricting `onInstall`/`onUninstall`, as every 7579 compliant fallback will HAVE to protect those methods, by requiring 2771-sender being the EP or SA itself.

Spearbit: Acknowledged.

5.3.7 Execute functions may incorrectly report success for some calls

Severity: Low Risk

Context: ExecutionHelper.sol#L38-L43, ExecutionHelper.sol#L67

Description: All execution methods (except the delegatecalls) use either `_execute()` or `_tryExecute()` to make low-level calls to the target address with the supplied calldata. `_execute()` reverts on failure, and `_tryExecute()` returns the call success variable.

But both of these functions do not correctly mimic Solidity's behavior for external calls.

```
function _tryExecute(address target, uint256 value, bytes calldata callData) internal virtual returns
↪  (bool success, bytes memory result) {

    assembly {
        result := mload(0x40)
        calldatacopy(result, callData.offset, callData.length)

        success := call(gas(), target, value, result, callData.length, codesize(), 0x00)

        mstore(result, returndatasize()) // Store the length.
        let o := add(result, 0x20)
        returndatacopy(o, 0x00, returndatasize()) // Copy the returndata.
        mstore(0x40, add(o, returndatasize())) // Allocate the memory.

    }
```

The above snippet is from `_tryExecute()`. There is no consideration for low-level EVM calls returning call success as true when there is no code at the target address.

When this happens, the code should behave identically to Solidity. In the case of `_execute()`, it will not revert though the call is actually considered to be a failure as per solidity, and in the case of `_tryExecute()`, it will skip emitting the `TryExecuteUnsuccessful()` event.

Recommendation: Decide if calls to an EOA should be counted as successful or not, and document this behavior.

In case it should not be counted as a success, consider adding a check for `extcodesize` to the assembly block before the low-level call, and return call `success = false` if the target has no code.

Biconomy: Acknowledged. I think it would be best to only document what is expected behavior.

Spearbit: Acknowledged.

5.3.8 Events not emitted for failed `tryExecute` attempts

Severity: Low Risk

Context: ExecutionHelper.sol#L143, ExecutionHelper.sol#L163

Description: When calling `excecute()` with `EXECTYPE_TRY` and either `CALLTYPE_SINGLE` or `CALLTYPE_DELEGATECALL`, no event is emitted if the try fails.

Whereas with `CALLTYPE_BATCH`, the `TryExecuteUnsuccessful` event is emitted for each failure.

In `executeFromExecutor` `TryDelegateCallUnsuccessful` and `TryExecuteUnsuccessful` events are emitted appropriately.

Recommendation: Update code to emit events consistently:

```
- else if (execType == EXECTYPE_TRY) _tryExecute(target, value, callData);
+ else if (execType == EXECTYPE_TRY) {
+     (bool success, bytes memory data) = _tryExecute(target, value, callData);
+     if (!success) emit TryExecuteUnsuccessful(0, data);
+ }
```

```
- else if (execType == EXECTYPE_TRY) _tryExecuteDelegatecall(delegate, callData);
+ else if (execType == EXECTYPE_TRY) {
+     (bool success, bytes memory data) = _tryExecuteDelegatecall(delegate, callData);
+     if (!success) emit TryDelegateCallUnsuccessful(0, data);
+ }
```

Biconomy: Fixed in PR 127.

Spearbit: Fixed.

5.3.9 Enable Mode can bypass the registry on multi-type validator modules

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: A validator that does not meet the minimum threshold of attestors in the ERC-7484 registry can be used to pass validation through "*enable mode*" during `validateUserOp`.

Enable mode is intended as a feature whereby a new validator module can be installed and immediately used for validation of a user op when `validateUserOp` is called. Initially, the nonce is checked to see if enable mode has been requested, in which case the _enableMode internal function is called.

```
function validateUserOp(
    PackedUserOperation calldata op,
    bytes32 userOpHash,
    uint256 missingAccountFunds
) external virtual payPrefund(missingAccountFunds) onlyEntryPoint returns (uint256 validationData) {
    address validator = op.nonce.getValidator();
    if (!op.nonce.isModuleEnableMode()) {
        // Check if validator is not enabled. If not, return VALIDATION_FAILED.
        if (!_isValidatorInstalled(validator)) return VALIDATION_FAILED;
        validationData = IValidator(validator).validateUserOp(op, userOpHash);
    } else {
        PackedUserOperation memory userOp = op;
        userOp.signature = _enableMode(validator, op.signature);
        validationData = IValidator(validator).validateUserOp(userOp, userOpHash);
```

However, the enableMode function logic does not check if the type of module is a validator. Therefore, as long as the signature is valid, any type of module can be installed through enable mode, not just a validator type.

```
function _enableMode(address module, bytes calldata packedData) internal returns (bytes calldata
↪  userOpSignature) {
    uint256 moduleType;
    bytes calldata moduleInitData;
    bytes calldata enableModeSignature;

    (moduleType, moduleInitData, enableModeSignature, userOpSignature) =
    ↪  packedData.parseEnableModeData();
    _checkEnableModeSignature(
        _getEnableModeDataHash(module, moduleInitData),
        enableModeSignature
    );
    _installModule(moduleType, module, moduleInitData);
}
```

Upon installation of the new module, the control flow continues with the last line in validateUserOp() which calls the new module.

```
userOp.signature = _enableMode(validator, op.signature);
validationData = IValidator(validator).validateUserOp(userOp, userOpHash);
```

Multi-type modules can be both validators and other types of modules as well. Each type must be attested to on the 7484 registry. A multi-type module may have many attestors in one module-type and few in another.

For example, a multi-type module may meet the attestor threshold as a `MODULE_TYPE_EXECUTOR` module but not as a `MODULE_TYPE_VALIDATOR`. When this type of module is passed in enable mode with the type `EXECUTOR` selected, it will be successfully installed (as an executor) and then immediately used as a validator. Normally, upon installation of a new validator, there is a check made on the registry for the module address and module type `VALIDATOR`. But in this case, since it is being installed as an executor, it will not check the validator type.

Recommendation: Consider adding the same validator validation used above in the non-enable-mode case:

```
@@ -107,6 +107,7 @@ contract Nexus is INexus, BaseAccount, ExecutionHelper, ModuleManager, UUPSUpgra
  } else {
      PackedUserOperation memory userOp = op;
      userOp.signature = _enableMode(validator, op.signature);
+     if (!_isValidatorInstalled(validator)) return VALIDATION_FAILED;
      validationData = IValidator(validator).validateUserOp(userOp, userOpHash);
```

Alternatively, consider including the module type in the `enableMode` signature hash along with a check that the module is a validator type.

Biconomy: Fixed in PR 129 and PR 126.

Spearbit: Fixed.

5.3.10 `withdrawDepositTo` overwrites free memory pointer for OOG

Severity: Low Risk

Context: BaseAccount.sol#L84-L93

Description: In the assembly block in the `withdrawDepositTo` function, The `amount` argument is written to `0x34` for the call to the entry point. This overwrites the top 20 bytes of the `0x40` memory slot (the free memory pointer).

In the case of an unsuccessful call, the free memory ptr is then loaded to use as the offset for copying the return data. But now, with the upper bits overwritten, this is a very high number which will result in Out Of Gas revert due to quadratic memory expansion costs. Furthermore, the actual revert return data, if any, will not bubble up.

Recommendation: Assign the value of the free memory pointer at the beginning and use that later.

```
+ let freeMemPtr := mload(0x40) // Store the free memory pointer.
  mstore(0x14, to) // Store the `to` argument.
  mstore(0x34, amount) // Store the `amount` argument.
  mstore(0x00, 0x205c2878000000000000000000000000) // `withdrawTo(address,uint256)`.
  if iszero(call(gas(), entryPointAddress, 0, 0x10, 0x44, codesize(), 0x00)) {
-     returndatacopy(mload(0x40), 0x00, returndatasize())
-     revert(mload(0x40), returndatasize())
+     returndatacopy(freeMemPtr, 0x00, returndatasize())
+     revert(freeMemPtr, returndatasize())
  }
```

Alternatively you could also overwrite the `0x00` address since you're reverting anyways.

Biconomy: Fixed in PR 125.

Spearbit: Fixed.

5.3.11 `createAccount` actual salt not deterministic given parameters

Severity: Low Risk

Context: K1ValidatorFactory.sol#L89, NexusAccountFactory.sol#L52, RegistryFactory.sol#L114-L127

Description: The `createAccount(bytes calldata initData, bytes32 salt)` function computes the actual `create2` salt (`actualSalt`) as the hash of its `calldata()` skipping the 4-bytes selector. Note that this includes the offsets to the `initData` and `salt` parameters.

The `AccountCreated` event is emitted with the `initData` and `salt` parameters.

However, only knowing the `initData` and `salt` parameters used for the creation is not enough to uniquely reconstruct the actual salt (and thereby the actual address) because of the calldata offsets being included in the hash. A non-default ABI encoding will lead to the same parameters but a different address which can be confusing and result in errors.

Recommendation: The `(initData, salt)` parameters should create a unique hash. Consider hashing the `initData` contents followed by the `salt` parameter instead.

```
bytes32 actualSalt = keccak256(abi.encodePacked(initData, salt))
```

> For `K1ValidatorFactory`, the hash should be computed over all parameters `keccak256(abi.encodePacked(eoaOwner, index, attesters, threshold))`.

The `computeAccountAddress` function should then also be changed to use the same code.

Biconomy: Fixed in PR 124.

Spearbit: Fixed.

5.4 Gas Optimization

5.4.1 Unused return data fir successful calls

Severity: Gas Optimization

Context: ExecutionHelper.sol#L142-L143, ExecutionHelper.sol#L152-L153, ExecutionHelper.sol#L162-L163

Description: `_execute()`, `_tryExecute()`, `_executeBatch()`, `_tryExecuteBatch()`, `_executeDelegatecall()` and `_tryExecuteDelegatecall()` copy return and revert data to memory, which isn't used at all. You can create another version of these functions where you skip handling return data completely.

Recommendation: Consider using SafeCall.call() from OP codebase where you can skip copying return data entirely. Also consider the context of the finding *"Events not emitted for failed `tryExecute` attempts"* since that introduces reading the revert data to emit events.

Biconomy: Fixed in PR 197.

Spearbit: Fixed.

5.4.2 Inefficient signature validation

Severity: Gas Optimization

Context: K1Validator.sol#L55, K1Validator.sol#L87-L88

Description: `K1Validator.validateUserOp()` validates signature as both smart contract signature and ECDSA signature. However, doing both the checks twice (`isValidSignatureNow()` is called twice) can be gas-inefficient as both the functions first validates the signature assuming an EOA owner, and if that fails, uses ERC1271 to validate smart contract signature.

So consider an instance where the owner is a smart contract and just signs on `userOpHash`. The signature validation has to go through 4 steps (2 failed validations for `ECDSA.toEthSignedMessageHash(userOpHash)`, 1 failed validation for `userOpHash` assuming EOA owner and finally the successful validation). This wastes a lot of gas. Instead, you can do a quick check if `owner.code.length == 0` and then validate the signature either through ecrecover or ERC1271.

Although smart contract owner is not currently possible as `onInstall()` ensures that the owner doesn't have any code which means it's an EOA. (however, there's a chance that code is deployed later at the owner address).

Recommendation: Consider one of the following fixes:

- The name `K1Validator` suggests that the owner is supposed to be an EOA addresses which uses secp256k1 curve for ECDA signatures. In that case, just use ecrecover for signature validation.
- If you do want to keep smart contract owners, remove the code length check from `onUninstall()`. Then call only one validation method based on `owner.code.length == 0`.

Biconomy: Fixed in PR 196 and PR 198.

Spearbit: Fixed.

5.4.3 Consider flipping `if/else` condition to save gas

Severity: Gas Optimization

Context: Nexus.sol#L103

Description: `if` condition takes a `NOT` of a boolean variable. To save gas, the code blocks inside if/else can be flipped to save on this extra `NOT` operation.

Recommendation: Consider switching the `if/else` condition as well as the code blocks to be executed inside.

Biconomy: Fixed in PR 112.

Spearbit: Fixed.

5.4.4 `K1Validator.validateUserOp` can be optimized

Severity: Gas Optimization

Context: K1Validator.sol#L88

Description: `K1Validator.validateUserOp` uses the `isValidSignatureNow` function.

Recommendation: Consider using `isValidSignatureNowCalldata` instead to save on gas.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.4.5 `_uninstallValidator` can be optimized

Severity: Gas Optimization

Context: ModuleManager.sol#L223

Description: The `_uninstallValidator` function checks if the previous or the next element of the validator to be removed is a "real address", i.e., not the `SENTINEL` value.

Recommendation: We can optimize this by performing the removal (`pop`) first, and only afterwards checking if the linked list is empty.

> Note that there's a bug in this library when calling `popAll()`, it leaves the list in an *uninitialized* state (`alreadyInitialized() == false`) instead of an initialized but empty state. While the code is not using `popAll()`, we still recommend including this check by ensuring the `next` element is also not the zero element.

Then the non-empty and initialized list check can be written as:

```
// perform removal first
validators.pop(prev, validator);
// Sentinel pointing to itself means empty list, pointing to 0 means uninitialized
require(validators.getNext(address(0x01)) != address(0x01)) && validators.getNext(address(0x01)) !=
↪  address(0x00)), CannotRemoveLastValidator());
```

Biconomy: Fixed in PR 139.

Spearbit: Fixed. We have left a comment on the PR.

5.4.6 Validation mode can be extracted from nonce more efficiently

Severity: Gas Optimization

Context: NonceLib.sol#L27

Description: The 1-byte validation mode is extracted from nonce as `shr(248, shl(24, nonce))`.

Recommendation: Consider using `let vmode := byte(3, nonce)` instead.

Biconomy: Fixed in PR 144.

Spearbit: Fixed.

5.5 Informational

5.5.1 Enforce non-zero address for owner for `K1Validator`

Severity: Informational

Context: K1Validator.sol#L54-L56

Description: `K1Validator.onInstall()` doesn't enforce that the owner has to be non-zero address. For comparison, `transferOwnership()` reverts if the new owner is the zero address.

Recommendation: Revert in `onInstall()` if `newOwner` is `address(0)`.

Biconomy: Fixed in PR 199.

Spearbit: Fixed.

5.5.2 Update code to align with Solady's ERC-7739 latest reference implementation

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The code currently utilizes an outdated version of Solady's ERC-7739 reference implementation. This may lead to inconsistencies with the latest version, potential security vulnerabilities, or missed optimizations introduced in the updated implementation.

Recommendation: Update the copied code to the latest version of Solady's ERC-7739 reference implementation.

Biconomy: Fixed in PR 177.

Spearbit: Fixed.

5.5.3 `BaseAccount` incorrectly inherits from Storage contract

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `Storage` contract defines all storage variables to be used within the nexus smart account.

`Nexus` inherits from `ModuleManager` and `BaseAccount`, and both of them inherit from this storage contract. This creates a convoluted inheritance pattern.

`BaseAccount`/ `ModuleManager` is not intended to be used separately and thus only one of these needs inheritance from storage, for the nexus account to be able to access it.

Recommendation: Remove `Storage` contract from the inheritance declaration of `BaseAccount` contract.

Biconomy: Fixed in PR 182.

Spearbit: Fixed.

5.5.4 Separate event can be added to represent the situation of hook emergency-uninstall request getting reset

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `emergencyUninstallHook()` function is meant to place new uninstall requests as well as handle the requests placed earlier.

In the case that 3 days have passed since the request was first placed, the request gets reset and now the hook can only be uninstalled after one more day has passed.

```
function emergencyUninstallHook(address hook, bytes calldata deInitData) external payable
↪  onlyEntryPoint {
    AccountStorage storage accountStorage = _getAccountStorage();
    uint256 hookTimelock = accountStorage.emergencyUninstallTimelock[hook];

    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) {
        // if the timelock has been left for too long, reset it
        accountStorage.emergencyUninstallTimelock[hook] = block.timestamp;
        emit EmergencyHookUninstallRequest(hook, block.timestamp);
    } else if (block.timestamp >= hookTimelock + _EMERGENCY_TIMELOCK) {
        // if the timelock expired, clear it and uninstall the hook
        accountStorage.emergencyUninstallTimelock[hook] = 0;
        _uninstallHook(hook, deInitData);
        emit ModuleUninstalled(MODULE_TYPE_HOOK, hook);
    } else {
        // if the timelock is initiated but not expired, revert
        revert EmergencyTimeLockNotExpired();
    }
}
```

The event that is emitted in the first else clause here (ie. reset) is not that helpful as it is equivalent to the situation when a new request is just being placed.

Recommendation: It is recommended to add a new event `EmergencyHookUninstallRequestReset()` to better represent the action of resetting an already existing uninstall request.

Biconomy: Fixed in PR 175.

Speabit: Fixed.

5.5.5 Modifiers `onlyValidatorModule()` and `onlyAuthorized()` are unused

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Two new modifiers have been introduced to `ModuleManager` contract: `onlyValidatorModule()` and `onlyAuthorized()`.

These are not used in the current codebase and the team clarified that there was no intention to use them in near future.

Recommendation: Remove dead functionality for better code clarity.

Biconomy: Fixed in PR 176.

Spearbit: Fixed.

5.5.6 Space after comma in type hash preimage

Severity: Informational

Context: Constants.sol#L41

Description: Type hash preimages like the following usually don't have a space after argument separators (`,`):

```
bytes32 constant MODULE_ENABLE_MODE_TYPE_HASH = keccak256("ModuleEnableMode(address module, bytes32
↪  initDataHash)");
```

Recommendation: Update it as:

```
- bytes32 constant MODULE_ENABLE_MODE_TYPE_HASH = keccak256("ModuleEnableMode(address module, bytes32
↪  initDataHash)");
+ bytes32 constant MODULE_ENABLE_MODE_TYPE_HASH = keccak256("ModuleEnableMode(address module,bytes32
↪  initDataHash)");
```

Biconomy: Fixed PR 112.

Spearbit: Fixed.

5.5.7 Specify what functions should be hookable

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Not all `Nexus` functions come with the `withHook` modifier. For example, `executeUserOp`, `installModule`, and `uninstallModule` are not hookable while the latter two are hookable in the reference implementation.

Recommendation: Clarify the functions that should have a `withHook` modifier and the reason why some functions do not have it.

Biconomy: All functions are hookable now. See commit 79376bc4.

Spearbit: Fixed.

5.5.8 Potential for Unrecoverable Smart Accounts Due to Lack of Active Validators

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The Nexus design prioritizes modularity, flexibility and customizability. While this approach offers significant benefits, it also introduces a critical vulnerability: if an account ends up with no active validators, it may become unrecoverable.

Several scenarios could lead to this state:

- Account initialization without validators as noted in the issue `Nexus.initializeAccount` does not check that a validator module has been installed.
- Deactivation of all validators due ERC-7484 whitelist changes: the current solution for this is to skip the registry check upon validator usage which contradicts the ERC-7484 recommendations and is further discussed in the issue "ERC-7484 registry checks missing on calls to modules".
- User actions that alter the state of the module. For example see the issue "`newOwner` may not have a private key".

- Unforeseen external factors that may alter the state of the module and cause it to become disabled. We cannot predict every possible use case and the flexibility offered by Nexus also allows for the possibility of complex interactions which may result in unexpected disabling of a module.

Recommendation: Below are some potential solutions to consider, however each comes with its own trade-offs:

- Implement a built-in fallback validator: this is contrary to the ethos of modularity maximalism and likely would require storing a signer address somewhere either in state or as immutable.
- Incorporating a form of social recovery: this is a large feature that would require a lot of new complexity and work.
- Creating an emergency mechanism for adding validators: this may introduce new risk and complexity.

There is no obvious solution. This issue highlights the tension between the goals of full modularity and ensuring account recoverability.

Biconomy: Fixed in PR 139. During the nexus initialization and module uninstallation, we ensure that the validators are not missing

Spearbit: This is just one of the scenarios this issue talks about, we can set fixed to the other issue and acknowledge this one.

5.5.9 `newOwner` may not have a private key

Severity: Informational

Context: K1Validator.sol#L70

Description: `newOwner` has to be an EOA capable of ECDSA signing. `K1Validator.onInstall()` just checks that this address has no code. For a stronger verification, it can also verify that `newOwner` is capable of signing by verifying an EOA signature here.

In the worst case, any address with no code can be passed which bricks this validator later, or code can be deployed on this address after installation. Since `validateUserOp()` checks for ERC1271 style signatures as well, a smart contract owner will later be able to control this validator.

Recommendation: Verify an ECDSA signature where `newOwner` is a signer to ensure `newOwner` is an EOA address and has a private key.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.5.10 Security Model & Trust assumptions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The smart account cannot protect against a user installing a malicious module. Therefore, we consider attacks relating to these out of scope.

The smart account however enforces certain "roles" (module types) of the modules which can act as safeguards. For example, only an executor module can execute a transaction through `executeFromExecutor` (however a malicious install-type module could install an executor).

Recommendation: Consider properly documenting the security model and trust assumptions. Inform users not to install arbitrary modules and to use the registry for additional security.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.5.11 Fallback handler static-calls not 7579 compliant

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: EIP-7579 states for the fallback:

> MUST use `call` to invoke the fallback handler

The current code also allows a `staticcall` instead of a `call` to the fallback handler.

Recommendation: Think about if Nexus should be fully EIP-7579 compliant. Otherwise, document this non-compliant behavior.

Biconomy: Acknowledged. This will be updated on EIP-7579.

Spearbit: Acknowledged.

5.5.12 Missing `memory-safe` assembly annotation

Severity: Informational

Context: ExecLib.sol#L19, LocalCallDataParserLib.sol#L19

Description: Some assembly code blocks are memory-safe but are missing the `memory-safe` annotation. See the linked code blocks.

Recommendation: Consider adding the annotation:

```
assembly ("memory-safe") {
// ...
}
```

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.13 Lack of granular control over ERC-7484 registry checks

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Currently, the use of the 7484 registry is an all-or-nothing proposition. Either the registry is turned on and the `registry` variable is set to the correct address, or else that variable is set to the zero address indicating the user has opted out of using the registry.

Users who would like to use the registry for most modules, but want to integrate with one custom module are required to register the new custom module and obtain the requisite number of attestations in order to use it.

It may be difficult or time consuming to get the attestations so the user is tempted to "disable" the registry altogether by setting the registry to the zero address.

Ideally, the 7484 registry would be able to configure threshold limits on a per module basis but that is currently not possible.

Recommendation: A small change to the RegistryAdapter contract could enable a user to pick and choose which modules should be checked against the registry.

```
  abstract contract RegistryAdapter {
      IERC7484 public registry;
+     mapping(address => bool) public registryEnabled;

      event ERC7484RegistryConfigured(IERC7484 indexed registry);

@@ -34,10 +35,8 @@ abstract contract RegistryAdapter {
  function _checkRegistry(address module, uint256 moduleType) internal view {
-     IERC7484 moduleRegistry = registry;
-     if (address(moduleRegistry) != address(0)) {
-         // this will revert if attestations / threshold are not met
-         moduleRegistry.check(module, moduleType);
+     if (registryEnabled[module]) && address(registry) != address(0)) {
+         registry.check(module, moduleType);
      }
  }
```

This change, plus the addition of logic to set and unset `registryEnabled` would allow a user to enable or disable the use of the registry on a module per module basis.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.5.14 `transferOwnership` can be replaced with a two-step process

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `transferOwnership()` is a sensitive function as it hands over the ownership of the smart account to a new address. This means that after the transfer, the new owner will be in charge of signing transactions that have to be executed on the smart account.

A user may want to simply change to his own secondary address. There is a chance that they will lose access to the account forever, and it could even get into malicious hands. The smart account needs to be guarded against anything that could compromise its integrity.

Recommendation: To ensure that the user does not lose complete ownership of the account due to a mistake, consider using a two-step ownership transfer process.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.5.15 Emitting the execution array index in case of a call failure might not be useful

Severity: Informational

Context: ExecutionHelper.sol#L99

Description: In the case of batch executions with execution mode = try/catch, for all failed attempts, the `TryExecuteUnsuccessful()` event is emitted to mark that some calls failed.

But this event is not that useful because it just returns the index position of the call in the batch of executions that was passed into the calldata.

Since it correctly emits the returned data, it should also return the execution data, which could be handy while decoding and tracking off-chain.

Recommendation: Consider changing the definition of the `TryExecuteSuccessful()` event or defining a new one to incorporate the execution data of a call that failed.

Biconomy: Fixed in PR 127.

Speabit: Fixed. Event `TryExecuteUnsuccessful` was redefined to include execution calldata instead of the index.

5.5.16 `_SELF` variable remains unused in `Nexus`

Severity: Informational

Context: Nexus.sol#L67

Description: In `Nexus.sol`, a `_SELF` variable has been defined and set to `address(this)` in the constructor.

The intended use of this variable seems to be as an immutable identifier of the smart account, but its usage is missing in the codebase.

Recommendation: One probable use case of the variable might be in the `onlyEntryPointOrSelf()` modifier. Consider using it, or remove it if it is not intended to be used, for better code clarity.

Biconomy: Fixed in commit 84c09410.

Speabit: Fixed.

5.5.17 Unnecessary module type check on uninstall

Severity: Informational

Context: Nexus.sol#L202

Description: The `uninstallModule` function checks if the module to be uninstalled is still of the module type when it was installed. This check seems unnecessary for non-malicious modules. For malicious modules this doesn't provide any further protection either.

Recommendation: Consider removing this check or clarify the additional security guarantees it provides.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.18 Registry threshold might be unattainable by attesters

Severity: Informational

Context: RegistryFactory.sol#L52-L53

Description: The `RegistryFactory` defines `attesters` and a `threshold` that they need to reach. However, there is no check that the attesters can actually attain the threshold.

Recommendation: Consider checking that `threshold <= attesters.length` in the constructor. It can also be added to the `setThreshold`, `addAttester`, and `removeAttester` function.

Biconomy: Fixed in PR 139.

Spearbit: This has only been done in the `constructor`. Leaving it as acknowledged.

5.5.19 Payable versus non-payable functions

Severity: Informational

Context: BaseAccount.sol#L81, Nexus.sol#L168, Nexus.sol#L225

Description: We noted there are functions that were marked as `payable`, presumably to allow for a delegatecall.

However, we noted the following functions which it was not clear if they should be marked as `payable` or not:

- `withdrawDepositTo` marked as `payable`.
- `setRegistry` currently not marked as `payable`.
- `executeUserOp` currently marked as `payable` but entry point sends 0 value.

Recommendation: Update functions as appropriate and provide clear documentation so there is no question what it should be.

Biconomy: They are mostly marked payable to save gas.

Speabit: Acknowledged.

5.5.20 `Nexus.initializeAccount` does not check that a validator module has been installed

Severity: Informational

Context: Nexus.sol#L221

Description: It's important that a validator module is installed during the initialization phase.

Otherwise, no other modules can be installed as they all require to be validated by a validator module. For this reason, `initializeAccount` delegatecalls into a bootstrap contract. However, it is not checked if a validator was indeed installed.

> Note that `_uninstallValidator` also reverts if the last validator would be removed.

Recommendation: Consider checking that the `validator` linked list is non-empty after the bootstrap call. Otherwise, revert.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.21 Callers of `executeFromExecutor` cannot know what calls succeeded

Severity: Informational

Context: Nexus.sol#L140

Description: EIP-7579 defines the following `executeFromExecutor` function:

```
function executeFromExecutor(bytes32 mode, bytes calldata executionCalldata)
    external
    returns (bytes[] memory returnData);
```

Note that when the `mode` is a `TRY_*` mode the inner calls can revert. The function does not indicate what subcalls succeeded to the caller as it only returns a `bytes[] memory returnData` but not `bool[] memory success`.

Recommendation: As this function is defined by the EIP, its return values cannot be changed. Consider adding another function that returns the success state of the individual inner calls. Currently, it's impossible to get the inner call status on-chain. Off-chain, failed calls will emit `TryExecuteUnsuccessful` events.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.5.22 Document custom `userOp.signature` encoding for `enableMode`

Severity: Informational

Context: LocalCallDataParserLib.sol#L8

Description: The `parseEnableModeData` function decodes the extended `userOp.signature` into `(ModuleType moduleType, bytes moduleInitData, bytes enableModeSignature, bytes userOpSignature)`. The encoding must follow a specially packed encoding as follows:

- First 32 bytes: `ModuleType moduleType` as a `uint256`.
- Next 4 bytes: Length of `moduleInitData`.
- Next <length-of-`moduleInitData`> bytes: `moduleInitData`.

- Next 4 bytes: Length of `enableModeSignature`.
- Next <length-of-`enableModeSignature`> bytes: `enableModeSignature`.
- Rest of the bytes: `userOpSignature`.

In Solidity, it would be encoded as follows:

```
bytes memory extendedUserOpSignature = abi.encodePacked(
    uint256(moduleType),
    bytes4(uint32(moduleInitData.length)),
    moduleInitData,
    bytes4(uint32(enableModeSignature.length)),
    enableModeSignature,
    userOpSignature
);
```

Recommendation: Document this custom encoding so integrators can use it. Consider also documenting it in the `parseEnableModeData` function.

Biconomy: Fixed in PR 139 and documented in the wiki.

Spearbit: Fixed.

5.5.23 Typos & Documentation errors

Severity: Informational

Context: IStorage.sol#L43, K1Validator.sol#L25, LocalCallDataParserLib.sol#L35, ModuleManager.sol#L315, ModuleManager.sol#L361, Nexus.sol#L327, Nexus.sol#L81, RegistryAdapter.sol#L34

Description: There are typos and documentation errors in the code:

1. LocalCallDataParserLib.sol#L35: `isntall` → `install`.
2. ModuleManager.sol#L361: Hooks are always global, there is no signature specific hook.
3. ModuleManager.sol#L315: This is a bit confusing, the first time I read it I thought the intention was to batch install modules. Something like "*This function installs multi-type modules, iterating through the different types and installing each type as a separate module*":

   ```
   /// To make it easier to install multiple modules at once, this function will
   ```

4. Nexus.sol#L81: This was confusing to read, I think it was copy pasted from a different part of the code and it doesn't actually apply here:

   ```
   /// The entryPoint calls this only if validation succeeds. Fails by returning VALIDATION_FAILED
   ↪  for invalid signatures.
   ```

5. Nexus.sol#L327: `fron` → `from`.
6. RegistryAdapter.sol#L34: `succicient` → 'sufficient'.
7. IStorage.sol#L43: Only `staticcall` and `call` are allowed, not `delegatecall`.
8. K1Validator.sol#L25: After talking with the team, this contract is supposed to be used in production, not only for testing purposes.

Recommendation: Consider addressing the mentioned issues.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.24 `ModuleType` type range should be restricted

Severity: Informational

Context: ModuleTypeLib.sol#L6

Description: The `ModuleType` type is used as an index in a 256-bit mask.

Therefore, its valid range is restricted to `[0, 255]` (inclusive). However, it is defined as a `uint256` and allows creating this type with larger values which will revert when used in the `isType` or `bitEncode*` functions.

Recommendation: Consider redefining the type to wrap a `uint8`, or create a `toModuleType(uint256 t)` function that checks that the given integer is in the valid range before constructing such a type.

```
function toModuleType(uint256 t) internal pure returns (ModuleType) {
    if (t > 255) revert InvalidModuleType(t);
    return ModuleType.wrap(uint256(t));
}
```

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.25 Undocumented `CALLTYPE_STATIC` call type

Severity: Informational

Context: ModeLib.sol#L68

Description: The `CALLTYPE_STATIC` (value `0xFE`) is not documented in EIP-7579 and is a new feature of Nexus. The fallback handlers can currently be configured to use this call type.

Recommendation: Consider documenting this new call type.

Biconomy: Acknowledged.

Spearbit: Acknowledged.

5.5.26 Missing `owner` address validation in `K1ValidatorFactory`

Severity: Informational

Context: K1ValidatorFactory.sol#L63

Description: All factories besides `K1ValidatorFactory` check that the `owner` is non-zero.

Recommendation: Check `factoryOwner != address(0)` in the constructor for consistency with other factories.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.27 Missing return parameters in natspec

Severity: Informational

Context: ExecutionHelper.sol#L104, ExecutionHelper.sol#L123, ExecutionHelper.sol#L170, ExecutionHelper.sol#L188, ExecutionHelper.sol#L198

Description: The definition of the return parameters is missing in natspec comments in the instances linked.

Recommendation: Consider adding descriptions about returned values to these functions, and completing natspec comments wherever required to enhance code readability.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.

5.5.28 `RegistryFactory.createAccount` assumes `Bootstrap.initNexus` will be used

Severity: Informational

Context: RegistryFactory.sol#L79

Description: The `RegistryFactory.createAccount` takes an `initData` parameter. This `initData` will be used to initialize the deployed Nexus with an `INexus(account).initializeAccount(initData);` call. This call then further delegatecalls into a `Bootstrap` contract function:

```
function initializeAccount(bytes calldata initData) external payable virtual {
    _initModuleManager();
    (address bootstrap, bytes memory bootstrapCall) = abi.decode(initData, (address, bytes));
    (bool success, ) = bootstrap.delegatecall(bootstrapCall);
    require(success, NexusInitializationFailed());
}
```

Note that `Nexus.initializeAccount` is opaque to what function to call on the `bootstrap` contract, however, the `RegistryFactory` assumes it will be calling a function of the `Bootstrap.initNexus` type, as it already decodes the `initData` according to it.

Recommendation: Document that `RegistryFactory.createAccount` is only compatible with an inner `Bootstrap.initNexus` call as its initialization method.

Biconomy: Fixed in PR 139.

Spearbit: Fixed.