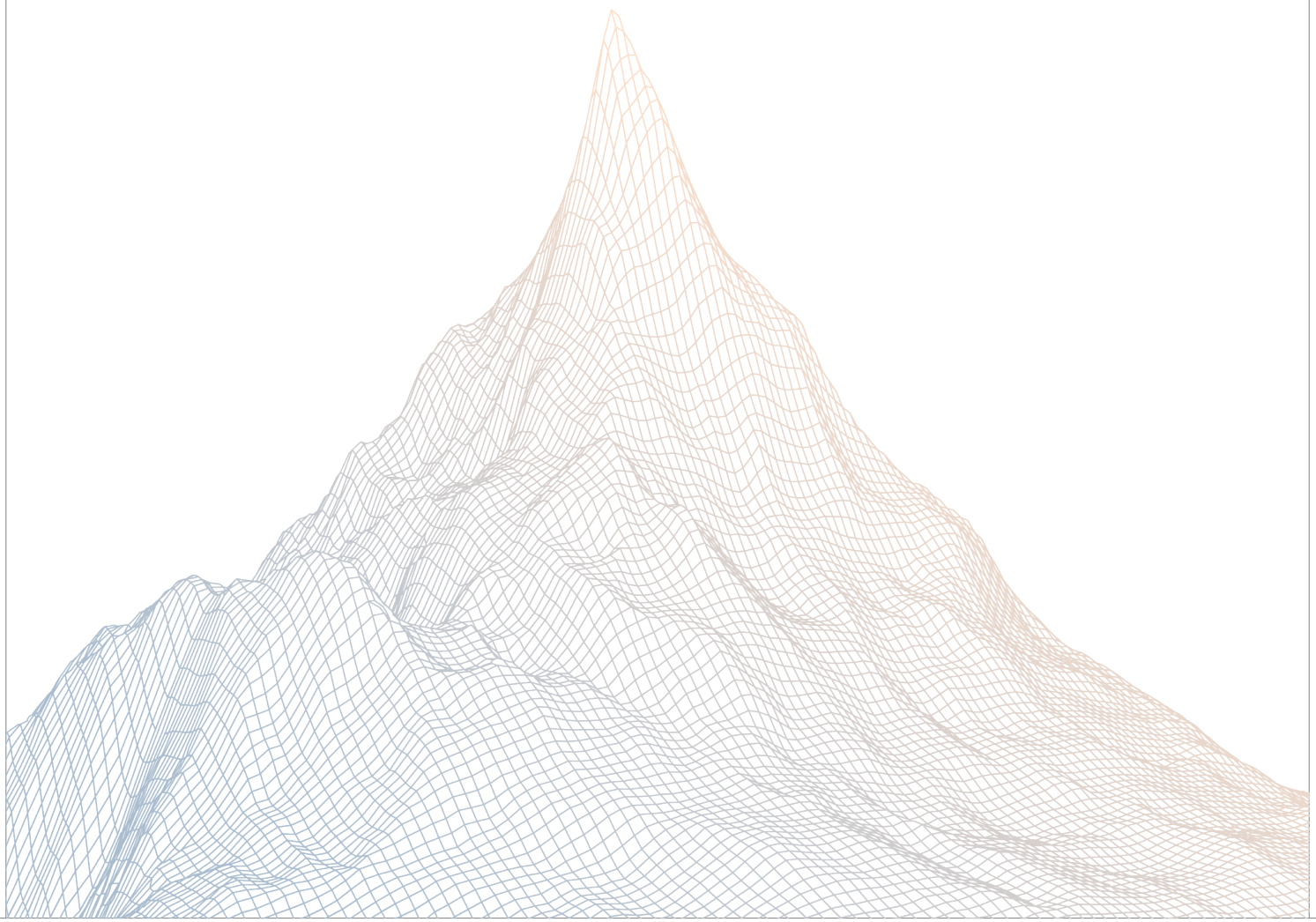
Zenith

Biconomy Nexus

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 10th to March 14th, 2025

AUDITED BY: cccz
chinmay

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Biconomy Nexus

Biconomy Nexus is building modular smart account infrastructure to simplify web3 interactions. This audit focuses on reviewing the changes introduced in PR-242, which implements updates to the account abstraction system following the ERC-4337 standard.

The scope of this review specifically covers the differential changes in PR-242, examining modifications to the account validation, module management, and security mechanisms.

2.2 Scope

The engagement involved a review of the following targets:

Target	nexus
Repository	https://github.com/bcnmy/nexus
Commit Hash	`e9cbc56dbec6b662248be52ed9e26e881f9a5665`
Files	`Diff on PR-242`

2.3 Audit Timeline

March 10, 2025	Audit start
March 14, 2025	Audit end
March 19, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	3
Low Risk	5
Informational	2
Total Issues	**11**

3

Findings Summary

ID	Description	Status
H-1	K1Validator.validateUserOp() validation can be bypassed	Resolved
M-1	emergencyUninstallHook() does not support DEFAULT_VALIDATOR	Resolved
M-2	NexusProxy might not be able to receive ether	Resolved
M-3	ERC7739 support detection logic excludes DEFAULT_VALIDATOR	Resolved
L-1	Initialization of DEFAULT_VALIDATOR is not mandated before using it for validation	Resolved
L-2	There is no way to initialize a Nexus account along with other modules	Resolved
L-3	The check to determine if the account is ERC7702 may not work correctly	Resolved
L-4	Uninstalling a pre-validation hook calls onUninstall() twice	Resolved
L-5	UserOp in Prep Mode cannot use the default validator	Resolved
I-1	Incomplete natspec docs for _installModule()	Resolved
I-2	Missing event emission in initNexusWithDefaultValidator()	Acknowledged

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] `K1Validator.validateUserOp()` validation can be bypassed

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- K1Validator.sol#L141-L143

Description:

`tryRecoverCalldata()` returns 0 on recovery failure, which is the same return value for uninitialized `smartAccountOwners`.

```
    // WARNING!
    // These functions will NOT revert upon recovery failure.
    // Instead, they will return the zero address upon recovery failure.
    // It is critical that the returned address is NEVER compared against
    // a zero address (e.g. an uninitialized address variable).

    /// @dev Recovers the signer's address from a message digest `hash`, and
    the `signature`.
    function tryRecover(bytes32 hash, bytes memory signature)
...
    /// @dev Recovers the signer's address from a message digest `hash`, and
    the `signature`.
    function tryRecoverCalldata(bytes32 hash, bytes calldata signature)
```

In `K1Validator.validateUserOp()`, an attacker can provide a malformed signature to bypass the validation.

```
function validateUserOp(PackedUserOperation calldata userOp,
    bytes32 userOpHash) external view override returns (uint256) {
    return _validateSignatureForOwner(smartAccountOwners[userOp.sender],
```

```
    userOpHash, userOp.signature) ? VALIDATION_SUCCESS : VALIDATION_FAILED;
}
```

Making this issue worse is that it is not necessary for Nexus to install the default validator.

This is because the default validator does not need to be installed to work, and will not be installed as long as the user does not call `NexusBootstrap.initNexusWithDefaultValidator()` in `initializeAccount()` when creating the Nexus.(`deployProxy()` only copies the _DEFAULT_VALIDATOR in the implementation bytecode and does not execute any `onInstall()` in the constructor.)

```
    function deployProxy(address implementation, bytes32 salt,
    bytes memory initData) internal returns (bool alreadyDeployed,
    address payable account) {
        // Check if the contract is already deployed
        account = predictProxyAddress(implementation, salt, initData);
        alreadyDeployed = account.code.length > 0;
        // Deploy a new contract if it is not already deployed
        if (!alreadyDeployed) {
            // Deploy the contract
            new NexusProxy{ salt: salt, value: msg.value }(implementation,
    abi.encodeCall(INexus.initializeAccount, initData));
...
    function initNexusWithDefaultValidator(
        bytes calldata data
    )
        external
        payable
    {
        IModule(_DEFAULT_VALIDATOR).onInstall(data);
    }
```

When the default validator is K1Validator, smartAccountOwners[user] may be address(0).

In `Nexus.validateUserOp()`, if calling the default validator, it is also not required that the default validator is installed.

```
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.isDefaultValidatorMode()) {
        validator = _DEFAULT_VALIDATOR;
    } else {
        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);
            _initializeAccount(initData);
        }
        validator = op.nonce.getValidator();
        require(_isValidatorInstalled(validator),
    ValidatorNotInstalled(validator));
    }

    (userOpHash, userOp.signature) = _withPreValidationHook(userOpHash,
    userOp, missingAccountFunds);
    validationData = IValidator(validator).validateUserOp(userOp,
    userOpHash);
}
```

This results in signature forging attacks on Nexus that do not have the default validator installed, but use it.

The POC is as follows, `DEFAULT_VALIDATOR_MODULE.smartAccountOwners(accountAddress)` is `address(0)` by default.

```
function test_poc() public payable {
    BootstrapConfig[] memory validators
    = BootstrapLib.createArrayConfig(address(VALIDATOR_MODULE), initData);
    BootstrapConfig[] memory executors
```

```
    = BootstrapLib.createArrayConfig(address(EXECUTOR_MODULE), "");
    BootstrapConfig memory hook
    = BootstrapLib.createSingleConfig(address(HOOK_MODULE), "");
    BootstrapConfig[] memory fallbacks
    = BootstrapLib.createArrayConfig(address(HANDLER_MODULE),
    abi.encode(bytes4(GENERIC_FALLBACK_SELECTOR)));
    bytes memory saDeploymentIndex = "0";
    bytes32 salt = keccak256(saDeploymentIndex);

    bytes memory _initData = BOOTSTRAPPER.getInitNexusCalldata(validators,
    executors, hook, fallbacks, registry, attesters, threshold);
    address alice = vm.addr(0x1234);
    vm.deal(alice, 1 ether);
    vm.prank(alice);
    address payable accountAddress = registryFactory.createAccount{ value:
    1 ether }(_initData, salt);

    console.log(DEFAULT_VALIDATOR_MODULE.smartAccountOwners(accountAddress));
    // 0x0000....
}
```

Recommendations:

```
    function validateUserOp(PackedUserOperation calldata userOp,
    bytes32 userOpHash) external view override returns (uint256) {

    return _validateSignatureForOwner(smartAccountOwners[userOp.sender],
           userOpHash, userOp.signature) ? VALIDATION_SUCCESS : VALIDATION_
           FAILED;

    return _validateSignatureForOwner(getOwner(userOp.sender), userOpHash,
           userOp.signature) ? VALIDATION_SUCCESS : VALIDATION_FAILED;
    }
```

And always enforce a DEFAULT_VALIDATOR isInitialized check everywhere the user tries to use it for anything

For both flows : EOA designating to Nexus, and the normal NexusProxy

Biconomy: Resolved with PR-254

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] `emergencyUninstallHook()` does not support `DEFAULT_VALIDATOR`

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ModuleManager.sol#L415-L426

Description:

`emergencyUninstallHook()` is used to emergency uninstall the Hook module, and in `_checkEmergencyUninstallSignature()` it will call `validator.isValidSignatureWithSender()` to validate the signature provided by the user.

```
    function emergencyUninstallHook(EmergencyUninstall calldata data,
    bytes calldata signature) external payable {
        // Validate the signature
        _checkEmergencyUninstallSignature(data, signature);
...
    function _checkEmergencyUninstallSignature(EmergencyUninstall calldata
    data, bytes calldata signature) internal {
        address validator = address(bytes20(signature[0:20]));
        require(_isValidatorInstalled(validator),
    ValidatorNotInstalled(validator));
        // Hash the data
        bytes32 hash = _getEmergencyUninstallDataHash(data.hook,
    data.hookType, data.deInitData, data.nonce);
        // Check if nonce is valid
        require(!_getAccountStorage().nonces[data.nonce], InvalidNonce());
        // Mark nonce as used
        _getAccountStorage().nonces[data.nonce] = true;
        // Check if the signature is valid
```

```
    require((IValidator(validator).isValidSignatureWithSender(address(this),
    hash, signature[20:]) == ERC1271_MAGICVALUE),
    EmergencyUninstallSigError());
    }
```

The problem here is that `_checkEmergencyUninstallSignature()` doesn't support the default validator, it requires that the validator be included in the account storage's validators, but the default validator isn't included.

```
function _isValidatorInstalled(address validator)
    internal view virtual returns (bool) {
    return _getAccountStorage().validators.contains(validator);
}
```

Recommendations:

```
    function _checkEmergencyUninstallSignature(EmergencyUninstall calldata
    data, bytes calldata signature) internal {
        address validator = address(bytes20(signature[0:20]));
        require(_isValidatorInstalled(validator), ValidatorNotInstalled(
            validator));
        address validator = _handleSigValidator(address(bytes20(signature[0:
            20])));
        // Hash the data
        bytes32 hash = _getEmergencyUninstallDataHash(data.hook,
    data.hookType, data.deInitData, data.nonce);
        // Check if nonce is valid
        require(!_getAccountStorage().nonces[data.nonce], InvalidNonce());
        // Mark nonce as used
        _getAccountStorage().nonces[data.nonce] = true;
        // Check if the signature is valid

    require((IValidator(validator).isValidSignatureWithSender(address(this),
    hash, signature[20:]) == ERC1271_MAGICVALUE),
    EmergencyUninstallSigError());
    }
```

And if the DEFAULT_VALIDATOR has to be used for validating this, it needs to be already initialized.

Biconomy: Resolved with PR-254

Zenith: Verified.

[M-2] NexusProxy might not be able to receive ether

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- NexusProxy.sol

Description:

When a new smart account is created from one of the Nexus factories, a `NexusProxy` contract gets deployed with implementation set to the Nexus contract. NexusProxy inherits functionality from the Openzeppelin Proxy contract.

The problem is that Openzeppelin Proxy contract does not have a receive function, as it delegates all calls to the payable fallback function.

If someone sends ether to the smart acocunt (ie. NexusProxy), its fallback function will be triggered. This may not work in case the external ETH sender transfers ETH via solidity's transfer method because it has a gas limit of 2300 while the NexusProxy fallback will use much more gas in some cases as it delegatecalls to the implementation.

References : Receiving-eth-in-proxy-contracts

How to receive ETH to an OpenZeppelin proxy contract

The smart account should be able to receive ether in all cases as its a user account, including direct ether transfers from external contracts and from within any executions that the account handles.

Recommendations:

Add a `receive()` function to `NexusProxy` so that ether transfers do not get forwarded to the delegate functionality.

Biconomy: Resolved with PR-256

Zenith: Verified.

[M-3] ERC7739 support detection logic excludes DEFAULT_VALIDATOR

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Nexus.sol

Description:

`isValidSignature()` ⇒ `checkERC7739Support()` flow in Nexus.sol is meant to determine if the smart account supports ERC7739. Nexus does so by looping over all the installed validators and returning true if any of the validators shows support for 7739 flows.

This is the `checkERC7739Support()` logic :

```
function checkERC7739Support(bytes32 hash, bytes calldata signature)
    public view virtual returns (bytes4) {
        bytes4 result;
        unchecked {
            SentinelListLib.SentinelList storage validators
    = _getAccountStorage().validators;
            address next = validators.entries[SENTINEL];
            while (next ≠ ZERO_ADDRESS && next ≠ SENTINEL) {
                bytes4 support
    = IValidator(next).isValidSignatureWithSender(msg.sender, hash,
    signature);
                if (bytes2(support) == bytes2(SUPPORTS_ERC7739) && support >
    result) {
                    result = support;
                }
                next = validators.getNext(next);
            }
        }
        return result == bytes4(0) ? bytes4(0xffffffff) : result;
    }
```

The problem here is that this code does not check the `DEFAULT_VALIDATOR` to find out if it supports 7739. This is because the `DEFAULT_VALIDATOR` is not included in the validators

storage.

As a consequence, for some smart accounts that only have DEFAULT_VALIDATOR enabled, ERC7739 support detection requests will return false even though the DEFAULT_VALIDATOR may support ERC7739.

Recommendations:

After the loop ends, the `DEFAULT_VALIDATOR` should also be checked and the returned bytes4 value should be intersected with the loop's outcome to clearly portray ERC7739 support.

Biconomy: Resolved with PR-254

Zenith: Verified

4.3 Low Risk

A total of 5 low risk findings were identified.

[L-1] Initialization of DEFAULT_VALIDATOR is not mandated before using it for validation

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Nexus.sol
- ModuleMaanager.sol#L610-L617

Description:

At many places, it is allowed to use the `DEFAULT_VALIDATOR` for validating a signature, but it is not guaranteed that the `DEFAULT_VALIDATOR` is indeed installed on the smart account.

This happens at following instances :

- If used as `enableModeSigValidator` in `enableMode`
- In `defaultValidator` Mode
- In prep mode, if the recommendation in issue zenith-security/2025-03-biconomy-nexus#9 is followed
- While checking the emergency Uninstall signature, if recommendation in zenith-security/2025-03-biconomy-nexus#11 is followed

This problem arises because many init methods in NexusBootstrap do not enforce `onInstall()` call for DEFAULT_VALIDATOR.

This could lead to a non-configured validator being used to verify signatures, eventually resulting in unexpected scenarios like the one mentioned in issue zenith-security/2025-03-biconomy-nexus#12.

Recommendations:

It is recommended to check that the `DEFAULT_VALIDATOR` has been initialized, at all places a user tries to use it for verifying signatures. This can be done by checking

`DEFAULT_VALIDATOR.isInitialized()` flag in `_handleSigValidator()`.

Biconomy: I think the check for Default Validator to be initialized is not required. Because, for example, K1 validator being used as default validator is always 'initialized' in a way. If the owner is address(0), it will try to use the smart account itself as a signer. For 7702 accounts, it is just easiest way

Since we control the default validator when deploying Nexus implementation and it can not be changed by the user, we can be sure that the validator that supports such a 'fallback' owner is always used as a default one

The only case when it actually makes sense to check if the default validator has been initialized is the case of a non-7702 account because it can not sign for itself. The check is introduced in PR-254

Zenith: Verified

[L-2] There is no way to initialize a Nexus account along with other modules

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- NexusBootstrap.sol

Description:

A nexus account cannot be initialized with a {DEFAULT_VALIDATOR + other modules} config. If they try to do so via `initNexus()` or other methods, the call will revert in `_installValidator()` because it prohibits from installing `DEFAULT_VALIDATOR` that way.

The user will have to separately initialize with `DEFAULT_VALIDATOR` and then later install the other required modules.

Recommendations:

Add a method to NexusBootstrap which can be used to initialize with {DEFAULT_VALIDATOR + other modules} config.

Biconomy: Resolved with PR-252

Zenith: Verified.

[L-3] The check to determine if the account is ERC7702 may not work correctly

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- ModuleManager.sol#L597-L607

Description:

The `_amIERC7702()` function performs some checks on the account's code to determine if it is an ERC7702 account. The code on the address needs to be 23 bytes and has to start with `0xef0100`.

```
function _amIERC7702() internal view returns (bool res) {
    assembly {
        // use extcodesize as the first cheapest check
        if eq(extcodesize(address()), 23) {
            // use extcodecopy to copy first 3 bytes of this contract and
    compare with 0xef0100
            extcodecopy(address(), 0, 0, 3)
            res := eq(0xef01, shr(240, mload(0x00)))
        }
        // if it is not 23, we do not even check the first 3 bytes
    }
}
```

This check is then used to prevent upgrading the account via the `UUPS.upgradeAndCall()` methods.

But the problem is that this code compares only first two bytes instead of 3 bytes.

This might create problems in future if `0xef01XX` is chosen for some other functionality.

Recommendations:

Compare all 3 bytes to prevent unexpected scenarios in future.

Biconomy: Resolved with PR-257

Zenith: Verified.

[L-4] Uninstalling a pre-validation hook calls `onUninstall()` twice

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- ModuleManager.sol#L264-L271
- ModuleManager.sol#L348-L351

Description:

When uninstalling a pre-validation hook from the Nexus account, `_uninstallHook()` is called.

```
function _uninstallHook(address hook, uint256 hookType, bytes calldata data)
    internal virtual {
        if (hookType == MODULE_TYPE_HOOK) {
            _setHook(address(0));
        } else if (hookType == MODULE_TYPE_PREVALIDATION_HOOK_ERC1271 ||
    hookType == MODULE_TYPE_PREVALIDATION_HOOK_ERC4337) {
            _uninstallPreValidationHook(hook, hookType, data);
        }
        hook.excessivelySafeCall(gasleft(), 0, 0,
    abi.encodeWithSelector(IModule.onUninstall.selector, data));
    }
```

And this is `_uninstallPreValidationHook()` code :

```
function _uninstallPreValidationHook(address preValidationHook,
    uint256 hookType, bytes calldata data) internal virtual {
    _setPreValidationHook(hookType, address(0));
    preValidationHook.excessivelySafeCall(gasleft(), 0, 0,
    abi.encodeWithSelector(IModule.onUninstall.selector, data));
}
```

As we can see, for prevalidation hooks, `onUninstall()` is called twice.

This can lead to problems and unexpected outcomes in future (potentially preventing

hooks from being uninstalled).

Calling onUninstall() twice is not needed.

Recommendations:

```
function _uninstallPreValidationHook(address preValidationHook,
    uint256 hookType, bytes calldata data) internal virtual {
        _setPreValidationHook(hookType, address(0));
-       preValidationHook.excessivelySafeCall(gasleft(), 0, 0,
    abi.encodeWithSelector(IModule.onUninstall.selector, data));
    }
```

Biconomy: Resolved with PR-257

Zenith: Verified

[L-5] UserOp in Prep Mode cannot use the default validator

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Nexus.sol#L131-L141

Description:

In PrepMode, `validateUserOp()` will first call `_initializeAccount()` to initialize the account, if there is no validator installed in `_initializeAccount()`, this means that the user will use the default validator, but for the default validator, `_ isValidatorInstalled()` will return false, which prevents userOp in PrepMode from using the default validator.

```
} 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);
    _initializeAccount(initData);
}
validator = op.nonce.getValidator();
require(_isValidatorInstalled(validator), ValidatorNotInstalled(validator));
```

Recommendations:

It is recommended that the default validator be allowed in PrepMode, and when using the default validator in PrepMode, it needs to be checked that the default validator has been initialized.

Biconomy: Resolved with PR-254

Zenith: Verified.

4.4 Informational

A total of 2 informational findings were identified.

[I-1] Incomplete natspec docs for `_installModule()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ModuleManager.sol#L171-L183

Description:

Changes have been made to natspec docs throughout the codebase to reflect two new types of modules (`MODULE_TYPE_PREVALIDATION_HOOK_ERC1271`, `MODULE_TYPE_PREVALIDATION_HOOK_ERC4337`), but the same modifications have not been made to natspec in `_installModule()`.

The description of these two new module types is missing from here :

```
/// @notice Installs a new module to the smart account.
    /// @param moduleTypeId The type identifier of the module being
    installed, which determines its role:
    /// - 0 for MultiType
    /// - 1 for Validator
    /// - 2 for Executor
    /// - 3 for Fallback // @audit missing docs for the 2 new types of
    pre-validation hooks.
    /// - 4 for Hook

...

    function _installModule(uint256 moduleTypeId, address module,
    bytes calldata initData) internal withHook {
```

Recommendations:

Add documentation for identifiers of these two new module types.

Biconomy: Resolved with PR-257

Zenith: Verified

[I-2] Missing event emission in `initNexusWithDefaultValidator()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- NexusBootstrap.sol#L53-L60

Description:

When a Nexus account is initialized using `initNexusWithDefaultValidator()`, the DEFAULT_VALIDATOR is installed by delegatecall from NexusProxy => Nexus imp => NexusBootstrap.sol

While all other methods of initializing an account emit ModuleInstalled events, this particular method lacks event emission even though a validator is being installed.

Recommendations:

Add `ModuleInstalled()` event emission for this function too.

Biconomy: Acknowledged