# Biconomy Audit Report

Prepared by CodeHawks

Version 1.0

September 16, 2024

# Contents

# 1 About CodeHawks

Codehawks helps protect some of the world's largest protocols. It offers private and public smart contract auditing competitions, supported by a global community of industry-leading security researchers. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at codehawks.cyfrin.io.

# 2 Disclaimer

The CodeHawks team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit contest does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

# 3 Risk Classification

| | **Impact: High** | **Impact: Medium** | **Impact: Low** |
|---|---|---|---|
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

# 4 Audit Scope

```
contracts/
 Nexus.sol
 base/
 common/
 factory/
 interfaces/
 lib/
 modules/
 types/
 utils/
```

# 5 Executive Summary

Over the course of July 8th - July 15th, CodeHawks auditors conducted an audit on the Biconomy smart contracts provided by Biconomy. In this period, a total of 14 issues were found.

**Summary**

| | |
|---|---|
| Project Name | Biconomy |
| Repository | 2024-07-biconomy |
| Audit Timeline | July 8th - July 15th |
| Methods | Competitive Audit |

**Issues Found**

| | |
|---|---|
| High Risk | 4 |
| Medium Risk | 4 |
| Low Risk | 6 |
| Total Issues | 14 |

# 6 Findings Overview

-

## 6.1 High Risk Findings

- H-01. User may lose funds when creating Nexus account or executing user operations
- H-02. Registry is never called when setting up modules using the `Bootstrap` contract
- H-03. Installing validators with enable mode in `validateUserOp()` doesn't check `moduleType`
- H-04. Missing nonce in `_getEnableModeDataHash()` allows signature replay

-

## 6.2 Medium Risk Findings

- M-01. Factory deployments won't work correctly on the ZKsync chain
- M-02. Anyone can call the fallbackFunction because of missing authorization control
- M-03. Protocol not fully compliant with `EIP-7579`
- M-04. Typehash for ModuleEnableMode struct is incorrect

-

## 6.3 Low Risk Findings

- L-01. Create account from `RegistryFactory` contract reverts due to unsorted external `attesters[]`
- L-02. The `indexed` Keyword in Events Causes Data Loss for Variables of type `bytes`
- L-03. entryPoint() function cannot be overridden
- L-04. `Nexus.validateUserOp()` violates the EIP-4337 specification
- L-05. Missing `_isInitialized(msg.sender)` check in `K1Validator.transferOwnership()`
- L-06. Fallback Handlers Can Be Installed with Invalid Calltype

# 7 High Risk Findings

## 7.1 H-01. User may lose funds when creating Nexus account or executing user operations

*Submitted by JuggerNaut63, y4y, sammy, Bauchibred, castleChain, 0xAbhayy, h2134, ke1caM, adriro, Sparrow, 0xRajkumar, MiloTruck, Shaheen. Selected submission by: h2134.*

### 7.1.1 Summary

User may lose funds when creating Nexus account or executing user operations, due to that ‘msg.value‘ is ignored.

### 7.1.2 Vulnerability Details

When user calls deployWithFactory() on `BiconomyMetaFactory` to create Nexus account, the `factory` contract specified by user will be called but `msg.value` is ignored:

```
    function deployWithFactory(address factory, bytes calldata factoryData) external payable returns
    ↪  (address payable createdAccount) {
        require(factoryWhitelist[address(factory)], FactoryNotWhitelisted());
@>      (bool success, bytes memory returnData) = factory.call(factoryData);

        // Check if the call was successful
        require(success, CallToDeployWithFactoryFailed());

        // Decode the returned address
        assembly {
            createdAccount := mload(add(returnData, 0x20))
        }
    }
```

When executeUserOp() is called by EntryPoint, protocol extracts inner call data from user operation and calls to the target contract with the decoded data, but `msg.value` is ignored:

```
    function executeUserOp(PackedUserOperation calldata userOp, bytes32) external payable virtual
    ↪  onlyEntryPoint {
        // Extract inner call data from user operation, skipping the first 4 bytes.
        bytes calldata innerCall = userOp.callData[4:];
        bytes memory innerCallRet = "";

        // Check and execute the inner call if data exists.
        if (innerCall.length > 0) {
            // Decode target address and call data from inner call.
            (address target, bytes memory data) = abi.decode(innerCall, (address, bytes));
            bool success;
            // Perform the call to the target contract with the decoded data.
@>          (success, innerCallRet) = target.call(data);
            // Ensure the call was successful.
            require(success, InnerCallFailed());
        }

        // Emit the Executed event with the user operation and inner call return data.
        emit Executed(userOp, innerCallRet);
    }
```

When fallback() is revoked, is `calltype` is `CALLTYPE_SINGLE`, a fallback handler will be called, but again `msg.value` is ignore:

```
        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, 0, 0, add(calldatasize(), 20), 0, 0)) {
                    returndatacopy(0, 0, returndatasize())
                    revert(0, returndatasize())
                }
                returndatacopy(0, 0, returndatasize())
                return(0, returndatasize())
            }
        }
```

### 7.1.3 Impact

1. User will lose funds if they call with ETH;
2. Transaction may revert due to `msg.value` is not forwarded.

### 7.1.4 Tools Used

Manual Review

### 7.1.5 Recommendations

Handle `msg.value` properly.

1. **BiconomyMetaFactory.sol**

```
    function deployWithFactory(address factory, bytes calldata factoryData) external payable returns
    ↪  (address payable createdAccount) {
        require(factoryWhitelist[address(factory)], FactoryNotWhitelisted());
-        (bool success, bytes memory returnData) = factory.call(factoryData);
+        (bool success, bytes memory returnData) = factory.call{value: msg.value}(factoryData);

        // Check if the call was successful
        require(success, CallToDeployWithFactoryFailed());

        // Decode the returned address
        assembly {
            createdAccount := mload(add(returnData, 0x20))
        }
    }
```

2. **Nexus.sol**

```
    function executeUserOp(PackedUserOperation calldata userOp, bytes32) external payable virtual
    ↪  onlyEntryPoint {
        // Extract inner call data from user operation, skipping the first 4 bytes.
        bytes calldata innerCall = userOp.callData[4:];
        bytes memory innerCallRet = "";

        // Check and execute the inner call if data exists.
        if (innerCall.length > 0) {
            // Decode target address and call data from inner call.
            (address target, bytes memory data) = abi.decode(innerCall, (address, bytes));
            bool success;
            // Perform the call to the target contract with the decoded data.
-            (success, innerCallRet) = target.call(data);
+            (success, innerCallRet) = target.call{value: msg.value}(data);
            // Ensure the call was successful.
            require(success, InnerCallFailed());
```

```
        }

        // Emit the Executed event with the user operation and inner call return data.
        emit Executed(userOp, innerCallRet);
    }
```

3. **ModuleManager.sol**

```
        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, 0, 0, add(calldatasize(), 20), 0, 0)) {
+                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())
            }
        }
```

## 7.2 H-02. Registry is never called when setting up modules using the `Bootstrap` contract

*Submitted by h2134, adriro, Sancybars, 0xRajkumar, MiloTruck. Selected submission by: MiloTruck.*

### 7.2.1 Summary

In the `Bootstrap` contract, the registry is never called as modules are installed before calling `_configureRegistry()`.

### 7.2.2 Vulnerability Details

According to EIP-7484, the module registry must be queried at least once before or during the transaction in which a module is installed:

> A Smart Account MUST implement the following Adapter functionality either natively in the account or as a module. This Adapter functionality MUST ensure that:
>
> - The Registry is queried about module A at least once before or during the transaction in which A is called for the first time.

However, when setting up modules and the registry for smart accounts through the `Bootstrap` contract, the registry is only configured after modules are installed.

Using `initNexusWithSingleValidator()` as example, `_configureRegistry()` is only called after the validator has been installed in `_installValidator()`:

RegistryBootstrap.sol#L38-L47

```
function initNexusWithSingleValidator(
    IModule validator,
    bytes calldata data,
    IERC7484 registry,
    address[] calldata attesters,
    uint8 threshold
```

```
) external {
    _installValidator(address(validator), data);
    _configureRegistry(registry, attesters, threshold);
}
```

As a result, when modules are installed through the `Bootstrap` contract, the registry is never called as `registry` in `RegistryAdapter` has not been set when the `withHook` modifier (which calls `_checkRegistry`) is reached:

RegistryAdapter.sol#L36-L42

```
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);
    }
}
```

Essentially, the order of operations in `initNexusWithSingleValidator()` is:

- Call `_installValidator()`:
  - In `withHook`, `registry == address(0)` so the registry is not called.
  - Install the validator, which calls `validator.onInstall()`.
- Call `_configureRegistry()`, which sets `registry` to the registry address.

Therefore, since the registry is never queried although `onInstall()` is called on the modules being installed, the function violates the EIP-7484 spec.

Note that this applies to `initNexus()` and `initNexusScoped()` as well.

#### 7.2.3 Impact

When setting up modules through functions in `Bootstrap`, it is possible for modules not registered in the registry to be installed, which is a bypass of access control.

#### 7.2.4 Recommendations

For all functions in the `Bootstrap` contract, consider calling `_configureRegistry()` before installing modules.

### 7.3 H-03. Installing validators with enable mode in `validateUserOp()` doesn't check `moduleType`

*Submitted by adriro, MiloTruck. Selected submission by: MiloTruck.*

#### 7.3.1 Summary

`_checkEnableModeSignature()` doesn't check that `moduleType` is permitted by the owner's signature.

### 7.3.2 Vulnerability Details

When Nexus account owners send a transaction with enable mode in `PackedUserOperation.nonce`, `validateUserOp()` calls `_enableMode()` to install the validator as a new module.

Nexus.sol#L108-L109

```
PackedUserOperation memory userOp = op;
userOp.signature = _enableMode(validator, op.signature);
```

The `moduleType` and `moduleInitData` of the validator to be installed is decoded from `PackedUserOperation.signature`:

ModuleManager.sol#L166-L171

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

_checkEnableModeSignature(
    _getEnableModeDataHash(module, moduleInitData),
    enableModeSignature
);
_installModule(moduleType, module, moduleInitData);
```

As seen from above, to ensure that the account owner has allowed the validator to be installed with `moduleInitData`, `module` and `moduleInitData` are hashed, and the hash is checked to be signed by the owner with `enableModeSignature`.

However, `moduleType` is not included in the hash, as seen in `_getEnableModeDataHash()`:

ModuleManager.sol#L388-L398

```
function _getEnableModeDataHash(address module, bytes calldata initData) internal view returns (bytes32
↪  digest) {
    digest = _hashTypedData(
        keccak256(
            abi.encode(
                MODULE_ENABLE_MODE_TYPE_HASH,
                module,
                keccak256(initData)
            )
        )
    );
}
```

This allows a malicious relayer/bundler to call `validateUserOp()` and specify `moduleType` as any module type. For example, instead of `MODULE_TYPE_VALIDATOR`, the attacker can specify it as `MODULE_TYPE_EXECUTOR`.

If the validator happens to be a multi-type module, this is problematic an attacker can install the validator with any module type, without the owner's permission.

### 7.3.3 Impact

An attacker can install validators through enable mode and `validateUserOp()` with any module type, without permission from the owner.

Depending on the module being installed, this can have drastic consequences on the account, with the highest impact being executor modules as they can `delegatecall`.

### 7.3.4 Recommendations

If `_enableMode()` is only meant to install validators, consider calling `_installModule()` with `MODULE_TYPE_VALIDATOR` instead of having it as a parameter.

Otherwise, include `moduleType` in the hash returned by `_getEnableModeDataHash()`. This ensures that the module type is permitted by the account owner's signature.

## 7.4 H-04. Missing nonce in `_getEnableModeDataHash()` allows signature replay

*Submitted by MiloTruck.*

### 7.4.1 Summary

`_getEnableModeDataHash()` doesn't include a nonce, thereby allowing enable mode signatures to be replayed.

### 7.4.2 Vulnerability Details

When Nexus account owners send a transaction with enable mode in `PackedUserOperation.nonce`, `validateUserOp()` calls `_enableMode()` to install the validator as a new module.

Nexus.sol#L108-L109

```
PackedUserOperation memory userOp = op;
userOp.signature = _enableMode(validator, op.signature);
```

To ensure that the account owner has allowed the validator to be installed, the validator (ie. `module` shown below) is hashed alongside its data (ie. `moduleInitData`) in `_getEnableModeDataHash()`, and subsequently checked to be signed by the owner in `enableModeSignature` in `_checkEnableModeSignature()`:

ModuleManager.sol#L166-L171

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

_checkEnableModeSignature(
    _getEnableModeDataHash(module, moduleInitData),
    enableModeSignature
);
_installModule(moduleType, module, moduleInitData);
```

However, the hash returned by `_getEnableModeDataHash()` does not include a nonce:

ModuleManager.sol#L388-L398

```
function _getEnableModeDataHash(address module, bytes calldata initData) internal view returns (bytes32
↪  digest) {
    digest = _hashTypedData(
        keccak256(
            abi.encode(
                MODULE_ENABLE_MODE_TYPE_HASH,
                module,
                keccak256(initData)
            )
```

```
        )
    );
}
```

This allows the owner's signature to be used repeatedly.

As a result, if a validator that was previously installed through `_enableMode()` is uninstalled by the owner, a malicious relayer/bundler can re-use the previous signature to re-install it through `validatorUserOp()` again, despite not having the owner's permission.

#### 7.4.3 Impact

Due to signature replay, validators that have been uninstalled by Nexus account owners can be re-installed without their permission.

This is especially problematic as validators are used by Nexus accounts for access control - being able to re-install a validator without the owner's permission might affect the Nexus account's permissions and allow attackers to execute transactions on behalf of the account.

#### 7.4.4 Recommendations

Include a nonce in `_getEnableModeDataHash()` to ensure that enable mode signatures cannot be replayed.

# 8 Medium Risk Findings

## 8.1 M-01. Factory deployments won't work correctly on the ZKsync chain

*Submitted by MSaptarshi007, sammy, 0xTheBlackPanther, Bauchibred, x18a6. Selected submission by: sammy.*

#### 8.1.1 Summary

Figure 1

#### 8.1.2 Vulnerability Details

The contract factories, namely `NexusAccountFactory.sol`, `RegistryFactory.sol`, and `RegistryFactory.sol` use Solady's `LibClone` library, which will not work correctly on the ZKsync chain. This is because, for the `create/create2` opcodes to function correctly on the ZKsync chain, the compiler must be aware of the bytecode of the deployed contract in advance.

Quoting from ZKsync docs :

"*On ZKsync Era, contract deployment is performed using the hash of the bytecode, and the factoryDeps field of EIP712 transactions contains the bytecode. The actual deployment occurs by providing the contract's hash to the ContractDeployer system contract.*
*To guarantee that create/create2 functions operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance. The compiler interprets the calldata arguments as incomplete input for ContractDeployer, as the remaining part is filled in by the compiler internally. The Yul datasize and dataoffset instructions have been adjusted to return the constant size and bytecode hash rather than the bytecode itself.*"

Let's look at how the contracts are cloned in the factory contracts :

NexusAccountFactory.sol#L56 :

```
    function createAccount(bytes calldata initData, bytes32 salt) external payable override returns
    ↪  (address payable) {
        // Compute the actual salt for deterministic deployment
```

```
        bytes32 actualSalt;
        assembly {
            let ptr := mload(0x40)
            let calldataLength := sub(calldatasize(), 0x04)
            mstore(0x40, add(ptr, calldataLength))
            calldatacopy(ptr, 0x04, calldataLength)
            actualSalt := keccak256(ptr, calldataLength)
        }


        // Deploy the account using the deterministic address
@=>         (bool alreadyDeployed, address account) = LibClone.createDeterministicERC1967(msg.value,
↪   ACCOUNT_IMPLEMENTATION, actualSalt);


        if (!alreadyDeployed) {
            INexus(account).initializeAccount(initData);
            emit AccountCreated(account, initData, salt);
        }
        return payable(account);
    }
```

RegistryFactory.sol#L123 :

```
            actualSalt := keccak256(ptr, calldataLength)
        }


        // Deploy the account using the deterministic address
@=>         (bool alreadyDeployed, address account) = LibClone.createDeterministicERC1967(msg.value,
↪   ACCOUNT_IMPLEMENTATION, actualSalt);


        if (!alreadyDeployed) {
            INexus(account).initializeAccount(initData);
            emit AccountCreated(account, initData, salt);
        }
```

K1ValidatorFactory.sol#L93 :

```
    function createAccount(
        address eoaOwner,
        uint256 index,
        address[] calldata attesters,
        uint8 threshold
    ) external payable returns (address payable) {
        // Compute the actual salt for deterministic deployment
        bytes32 actualSalt;
        assembly {
            let ptr := mload(0x40)
            let calldataLength := sub(calldatasize(), 0x04)
            mstore(0x40, add(ptr, calldataLength))
            calldatacopy(ptr, 0x04, calldataLength)
            actualSalt := keccak256(ptr, calldataLength)
        }

        // Deploy the Nexus contract using the computed salt
@=>         (bool alreadyDeployed, address account) = LibClone.createDeterministicERC1967(msg.value,
↪   ACCOUNT_IMPLEMENTATION, actualSalt);

        // Create the validator configuration using the Bootstrap library
```

```
        BootstrapConfig memory validator = BootstrapLib.createSingleConfig(K1_VALIDATOR,
        ↪  abi.encodePacked(eoaOwner));
        bytes memory initData = BOOTSTRAPPER.getInitNexusWithSingleValidatorCalldata(validator,
        ↪  REGISTRY, attesters, threshold);

        // Initialize the account if it was not already deployed
        if (!alreadyDeployed) {
            INexus(account).initializeAccount(initData);
            emit AccountCreated(account, eoaOwner, index);
        }
        return payable(account);
    }
```

They all use the `LibClone.createDeterministicERC1967()` method to create clones.

Now, let's look at the `LibClone.createDeterministicERC1967()` function :

```
    function createDeterministicERC1967(uint256 value, address implementation, bytes32 salt)
        internal
        returns (bool alreadyDeployed, address instance)
    {
        /// @solidity memory-safe-assembly
        assembly {
            let m := mload(0x40) // Cache the free memory pointer.
            mstore(0x60, 0xcc3735a920a3ca505d382bbc545af43d6000803e6038573d6000fd5b3d6000f3)
            mstore(0x40, 0x5155f3363d3d373d3d363d7f360894a13ba1a3210667c828492db98dca3e2076)
            mstore(0x20, 0x6009)
            mstore(0x1e, implementation)
            mstore(0x0a, 0x603d3d8160223d3973)
            // Compute and store the bytecode hash.
            mstore(add(m, 0x35), keccak256(0x21, 0x5f))
            mstore(m, shl(88, address()))
            mstore8(m, 0xff) // Write the prefix.
            mstore(add(m, 0x15), salt)
            instance := keccak256(m, 0x55)
            for {} 1 {} {
                if iszero(extcodesize(instance)) {
@=>                 instance := create2(value, 0x21, 0x5f, salt)
                    if iszero(instance) {
                        mstore(0x00, 0x30116425) // `DeploymentFailed()`.
                        revert(0x1c, 0x04)
                    }
                    break
                }
                alreadyDeployed := 1
                if iszero(value) { break }
                if iszero(call(gas(), instance, value, codesize(), 0x00, codesize(), 0x00)) {
                    mstore(0x00, 0xb12d13eb) // `ETHTransferFailed()`.
                    revert(0x1c, 0x04)
                }
                break
            }
            mstore(0x40, m) // Restore the free memory pointer.
            mstore(0x60, 0) // Restore the zero slot.
        }
    }
```

As you can see, the compiler will not be aware of the bytecode at compile time since the bytecode is stored in memory only on runtime in this function. The ZKsync docs recommend against this practice.

Since the compiler is unaware of the bytecode beforehand, this will lead to unexpected results on the ZKsync chain.

NOTE: This finding is heavily inspired by this issue

#### 8.1.3 Impact

Factory deployments will not work/ deploy faulty contracts on the ZKsync chain. Since the `createDeterministicERC1967()` method deploys ETH along with the contract, there will also be loss of funds.
Therefore, this is a High severity finding.

#### 8.1.4 Tools Used

Manual Review

#### 8.1.5 Recommendations

Make sure the compiler is aware of the bytecode beforehand.

Example (from ZKsync docs) :

```
bytes memory bytecode = type(MyContract).creationCode;
assembly {
    addr := create2(0, add(bytecode, 32), mload(bytecode), salt)
}
```

### 8.2 M-02. Anyone can call the fallbackFunction because of missing authorization control

*Submitted by BenRai, mxusee, adriro, zhuying, 0xAbhayy. Selected submission by: BenRai.*

#### 8.2.1 Lines of code

https://github.com/madeupnamefinance/2024-07-biconomy/blob/9590f25cd63f7ad2c54feb618036984774f3879d/contracts/base/ModuleManager.sol#L72C5-L110

#### 8.2.2 Impact

The lack of authorization control in the fallback function allows anyone to access and execute the fallback handlers. Depending on the fallback handler his can result in unauthorized transactions, data manipulation, or other unintended behaviours, potentially compromising the security and integrity of the smart account.

#### 8.2.3 Proof of Concept

1. An attacker sends a transaction to the `ModuleManager` contract with arbitrary data.
2. The fallback function is triggered due to the unrecognized function selector.
3. The fallback function routes the call to the corresponding fallback handler without verifying the sender's authorization.
4. The fallback handler executes the call, potentially leading to unauthorized actions.

#### 8.2.4 Recommended Mitigation Steps

Implement proper authorization control in the fallback function to ensure that only authorized entities can invoke it. This can be achieved by adding a modifier to check the sender's authorization before routing the call to the fallback handler. The existing onlyEntryPointOrSelf modifier could be used or an new modifier also including executorModuls might be appropriate.

### 8.3 M-03. Protocol not fully compliant with `EIP-7579`

*Submitted by BenRai, 0xTheBlackPanther, h2134, adriro, zhuying, 0xRajkumar, 0xShoonya, MiloTruck. Selected submission by: 0xShoonya.*

#### 8.3.1 Summary

The core smart contracts implementing modular smart accounts functionality in this protocol, specifically `Nexus.sol`, `ModuleManager.sol`, and `BaseAccount.sol`, do not include an implementation of `ERC-165` as required by `EIP-7579` specifications.

#### 8.3.2 Vulnerability Details

As per EIP-7579 specifications, smart accounts implementing this EIP must implement `EIP-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.

The `ERC-165` standard allows contracts to publish which interfaces they support, providing a way for contracts to query the support of interfaces without additional gas cost. The lack of `ERC-165` implementation means that the protocol is not fully compliant with `EIP-7579`.

#### 8.3.3 Impact

Medium. The protocol is not fully compliant with `EIP-7579`, which may lead to issues with interoperability and integration with other smart contracts and systems expecting `ERC-165` compliance.

#### 8.3.4 Tools Used

Manual Review

#### 8.3.5 Recommendations

To ensure full compliance with `EIP-7579`, implement `ERC-165`

## 8.4 M-04. Typehash for ModuleEnableMode struct is incorrect

*Submitted by adriro, ZanyBonzy, 0xShoonya, Fondevs. Selected submission by: adriro.*

### 8.4.1 Summary

There are two different errors in the typehash for the ModuleEnableMode structure, resulting in incompatibility with the EIP-712 standard.

### 8.4.2 Vulnerability Details

The typehash for the ModuleEnableMode structure is defined in the `MODULE_ENABLE_MODE_TYPE_HASH` constant as the following:

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

The first error is the blank space after the first comma between the members. According to the EIP-712 standard, members are joined by a "," without any blank space.

> The type of a struct is encoded as name ‖ "(" ‖ member ‖ "," ‖ member ‖ "," ‖ ... ‖ member ")" where each member is written as type ‖ " " ‖ name. For example, the above Mail struct is encoded as Mail(address from,address to,string contents).

The added space completely changes the result of the typehash:

```
keccak256("ModuleEnableMode(address module, bytes32 initDataHash)") =>
↪  0x9f34cc46003dee27a148c367555bac140aa3fbfc713cf44b3842f343fc2a4e7b
keccak256("ModuleEnableMode(address module,bytes32 initDataHash)")  =>
↪  0xe81b75314ee7a02a1f49592b5a66846e18745ee8948ef8f98f4649864a3f701e
```

There is another error in the data types, the typehash is used in the `_getEnableModeDataHash()` function:

```
388:     function _getEnableModeDataHash(address module, bytes calldata initData) internal view returns
↪  (bytes32 digest) {
389:         digest = _hashTypedData(
390:             keccak256(
391:                 abi.encode(
392:                     MODULE_ENABLE_MODE_TYPE_HASH,
393:                     module,
394:                     keccak256(initData)
395:                 )
396:             )
397:         );
398:     }
```

As we can see in the code snippet, the second argument, `initDataHash`, is of type `bytes` while the typehash defined it as `bytes32`.

### 8.4.3 Impact

Both errors will lead to an incompatibility with the EIP-712 standard. Existing tools and signing devices that implement EIP-712 will produce signatures that will be rejected due to these small errors even if the signer is valid.

### 8.4.4 Tools Used

None.

### 8.4.5 Recommendations

Change the typehash to:

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

# 9 Low Risk Findings

## 9.1 L-01. Create account from `RegistryFactory` contract reverts due to unsorted external `attesters[]`

*Submitted by BenRai, Tamer, TopStar, merlinboii, adriro, supplisec, sheep. Selected submission by: merlinboii.*

### 9.1.1 Summary

Unsorted external attesters cause the `RegistryFactory` contract to be unable to create the account, as it fails to validate the given module since the non-compliant `attesters[]` are used with the ERC-7484.

### 9.1.2 Vulnerability Details

The external `attesters[]` of the `RegistryFactory` contract are not guaranteed to be sorted when initialized in the constructor and are not sorted when the owner updates them via `addAttester()` and `removeAttester()`.

```
\Location: RegistryFactory.sol

    function addAttester(address attester) external onlyOwner {
@>          attesters.push(attester);  //  Potentially, it could not be sorted when added
    }

    function removeAttester(address attester) external onlyOwner {
        for (uint256 i = 0; i < attesters.length; i++) {
            if (attesters[i] == attester) {
@>              attesters[i] = attesters[attesters.length - 1];   // Potentially, it could not be sorted
↪  when removed
                attesters.pop();
                break;
            }
        }
    }
```

The external `REGISTRY` compliant with the ERC-7484 and used in the `isModuleAllowed()` function expects that the given external `attesters[]` are sorted to interact with the `REGISTRY.check()`.

> ERC-7484 Required Registry functionality:: `check` functions " The `attesters` provided **MUST** be unique and **sorted** and **the Registry MUST revert if they are not. "**

```
\Location: RegistryFactory.sol
```

```
    function isModuleAllowed(address module, uint256 moduleType) public view returns (bool) {
@>  REGISTRY.check(module, moduleType, attesters, threshold);
        return true;
    }
```

This unsorted `attesters[]` disrupts the `createAccount()` flow by reverting when invoking `isModuleAllowed()` to validate the given module and ensure all modules are whitelisted, resulting in the reversion in creating the `Nexus` account using the `RegistryFactory` contract.

```
\Location: RegistryFactory.sol

function createAccount(bytes calldata initData, bytes32 salt) external payable override returns
↪  (address payable) {

        // snipped

        // Ensure all modules are whitelisted
        for (uint256 i = 0; i < validators.length; i++) {
@>          require(isModuleAllowed(validators[i].module, MODULE_TYPE_VALIDATOR),
↪  ModuleNotWhitelisted(validators[i].module));
        }

        for (uint256 i = 0; i < executors.length; i++) {
@>          require(isModuleAllowed(executors[i].module, MODULE_TYPE_EXECUTOR),
↪  ModuleNotWhitelisted(executors[i].module));
        }

@>          require(isModuleAllowed(hook.module, MODULE_TYPE_HOOK), ModuleNotWhitelisted(hook.module));

        for (uint256 i = 0; i < fallbacks.length; i++) {
@>          require(isModuleAllowed(fallbacks[i].module, MODULE_TYPE_FALLBACK),
↪  ModuleNotWhitelisted(fallbacks[i].module));
        }

       // snipped

    }
```

### 9.1.3 Impact

The unsorted `attesters[]` can cause the `RegistryFactory` contract to revert when attempting to create a new account.

Although the **trusted** attesters are provided by the **owner** during the deployment phase, and the owner can compute and use sorted attester addresses before modification, the likelihood of this issue occurring is not considered low.

This is due to the arbitrary nature of address (20-bytes random) and the logic in the `addAttester()` and `removeAttester()` , which shows that there is no sorting logic present, making **it highly likely for the** `attesters[]` **array to become unsorted when modified**.

#### 9.1.4 Proof of Concept

**Initial State:**

- Using the `TrustManagerExternalAttesterList` contract as an example of the `REGISTRY` contract compliant with `ERC-7484`.

**Step 1**:

Deploy the `RegistryFactory` contract and assign the `attesters` as `[0x101, 0x202, 0x303, 0x404, 0x505]` (SORTED).

```
\Location: RegistryFactory.sol

    constructor(address implementation_, address owner_, IERC7484 registry_, address[] memory
    ↪  attesters_, uint8 threshold_) Stakeable(owner_) {
        require(implementation_ != address(0), ImplementationAddressCanNotBeZero());
        require(owner_ != address(0), ZeroAddressNotAllowed());
        REGISTRY = registry_;  //> TrustManagerExternalAttesterList
        attesters = attesters_;  //> [0x101, 0x202, 0x303, 0x404, 0x505]
        threshold = threshold_;
        ACCOUNT_IMPLEMENTATION = implementation_;
    }
```

**Step 2:**

Owner removes `0x101` attester => the new attesters become **UNSORTED **= `[0x505, 0x202, 0x303, 0x404]` (the unsorted logic also arises in the `addAttester()`).

```
\Location: RegistryFactory.sol

    function removeAttester(address attester: 0x101) external onlyOwner {
        for (uint256 i = 0; i < attesters.length; i++) {
            if (attesters[i] == attester) {
                attesters[i] = attesters[attesters.length - 1]; //> `0x505` wil replace the value at the
                ↪  index that stores `0x101`
                attesters.pop();
                break;
            }
        }
    }
```

**Step 3:**

User attempts to `createAccount()` but they obtain the revert as `REGISTRY.check(module, moduleType, attesters, threshold);` reverts.

```
\Location: RegistryFactory.sol

    function isModuleAllowed(address module, uint256 moduleType) public view returns (bool) {
 @> REGISTRY.check(module, moduleType, attesters, threshold);
        return true;
    }
```

```
\ TrustManagerExternalAttesterList.sol (example REGISTRY)

     /**
     * @inheritdoc IERC7484
     */
    function check(address module, ModuleType moduleType, address[] calldata attesters: `[0x505, 0x202,
    ↪  0x303, 0x404]`, uint256 threshold) external view {
        uint256 attestersLength = attesters.length;
        if (attestersLength == 0 || threshold == 0) {
```

```
            revert NoTrustedAttestersFound();
        } else if (attestersLength < threshold) {
            revert InsufficientAttestations();
        }

        address _attesterCache; // 0x000, used to store the PREV attester when loop
        for (uint256 i; i < attestersLength; ++i) {
            address attester = attesters[i]; //i_0, attester= attesters[0] = 0x505 || i_1,
            ↪  attester=attesters[1] = 0x202

@>          if (attester <= _attesterCache) revert InvalidTrustedAttesterInput(); //i_0, 0x505 <= 0x000
↪   (FALSE) || i_1, 0x202 <= 0x505 (TRUE) **REVERT**
            else _attesterCache = attester; //i_0, _attesterCache = 0x505 ||
            if ($getAttestation(module, attester).checkValid(moduleType)) {
                --threshold;
            }
            if (threshold == 0) return;
        }
        revert InsufficientAttestations();
    }
```

**Outcome:**

Temporary inability to create an account via the `RegistryFactory` contract until the owner modifies the attesters to be sorted via `addAttester()` and `removeAttester()`.

### 9.1.5 Tools Used

Manual Review

### 9.1.6 Recommendations

Ensure that the `attesters[]` array is sorted whenever it is modified. This can be achieved by implementing sorting logic in the `addAttester()` and `removeAttester()` or applyinh the `solady/LibSort`.

Moreover, applying the sorting logic in the `constructor` ensures that the `attesters` is sorted upon initialization.

```
\Location: RegistryFactory.sol

+import { LibSort } from "solady/src/utils/LibSort.sol";

contract RegistryFactory is Stakeable, INexusFactory {
+   using LibSort for address[];

// snipped


       function addAttester(address attester) external {
        attesters.push(attester);
+       address[] memory attesters_ = attesters;
+       attesters_.insertionSort();
+       attesters = attesters_;
    }

    function removeAttester(address attester) external {
        for (uint256 i = 0; i < attesters.length; i++) {
            if (attesters[i] == attester) {
                attesters[i] = attesters[attesters.length - 1];
                attesters.pop();
+               address[] memory attesters_ = attesters;
+               attesters_.insertionSort();
```

```
+                attesters = attesters_;
                 break;
             }
         }
     }

// snipped

}
```

## 9.2 L-02. The `indexed` Keyword in Events Causes Data Loss for Variables of type `bytes`

*Submitted by Matin.*

### 9.2.1 Summary

By making the event topic `indexed` for `bytes` type inside the `NexusAccountFactory` contract, it would lead for wrong variable to be emitted

### 9.2.2 Vulnerability Details

when the `indexed` keyword is used for reference type variables such as dynamic arrays or strings, it will return the hash of the mentioned variables. Thus, the event which is supposed to inform all of the applications subscribed to its emitting transaction (e.g. front-end of the DApp, or the backend listeners to that event), would get a meaningless and obscure 32 bytes that correspond to keccak256 of an encoded dynamic array. This may cause some problems on the DApp side and even lead to data loss. For more information about the indexed events, check here:

(https://docs.soliditylang.org/en/v0.8.17/abi-spec.html?highlight=indexed#events)

The problem exists inside the `NexusAccountFactory` contract. The event `AccountCreated` is defined in such a way that the `bytes` variable of `initData` is `indexed`. The function `createAccount()` is intended to create a new Nexus account with the provided initialization data. However, with the current design, the expected parameter wouldn't be emitted properly and front-end would get a meaningless one-way hash.

```
    /// @notice Emitted when a new Smart Account is created, capturing the account details and
    ↪  associated module configurations.
    event AccountCreated(address indexed account, bytes indexed initData, bytes32 indexed salt);
```

https://github.com/bcnmy/nexus/blob/main/contracts/interfaces/factory/INexusAccountFactory.sol#L27

### 9.2.3 Impact

### 9.2.4 Proof of Concept

Consider this scenario as an example:

1. The function `createAccount()` is called
2. Inside the function `createAccount()` we expect to see the the `bytes` of `initData` from calldata be emitted
3. But as the event topic is defined as `indexed` we'll get an obscure 32-byte hash and listeners will not be notified properly. Thus, the `initData` would be lost in the DApp.

For test purposes, one can run this test file:

```
    event AccountCreated(address indexed account, bytes indexed initData, bytes32 indexed salt);
    event AccountCreated1(address indexed account, bytes initData, bytes32 indexed salt);

    function test_emitter(bytes calldata initData) public {
```

```
        bytes32 salt = bytes32(abi.encodePacked(keccak256("N")));

        emit AccountCreated(address(this), initData, salt);
        emit AccountCreated1(address(this), initData, salt);
    }
```

Outputs of test: (with sample `initData` equal to `0x9cc7a4860f0b0926603c77f2e17ec5408745d45e2b668a287ed04e3aab0ea3d0`)

`AccountCreated` event:


```
[
    {
        "from": "0xa35EEbcac6c93ad12b4dedFB6E651c6eeB252541",
        "topic": "0x47e5b5fc3bda028416e26dcf66ca5186488c0717e8ab55bb01806113f1839d58",
        "event": "AccountCreated",
        "args": {
            "0": "0xa35EEbcac6c93ad12b4dedFB6E651c6eeB252541",
            "1": {
                "_isIndexed": true,
                "hash": "0x159308d99255c44d054988b9b6b1c977b00fd95c53a393332fde1842793b5dc8"
            },
            "2": "0x7c1e3133c5e040bb7fc55cda56e3c1998a2e33373c0850e92b53c932b65ceb44"
        }
    }
```


`AccountCreated1` event:


```
    {
        "from": "0xa35EEbcac6c93ad12b4dedFB6E651c6eeB252541",
        "topic": "0x211750db703759f2a69f94ed7881c771170ea46af0ddfc4bdb646b25638e9032",
        "event": "AccountCreated1",
        "args": {
            "0": "0xa35EEbcac6c93ad12b4dedFB6E651c6eeB252541",
            "1": "0x9cc7a4860f0b0926603c77f2e17ec5408745d45e2b668a287ed04e3aab0ea3d0",
            "2": "0x7c1e3133c5e040bb7fc55cda56e3c1998a2e33373c0850e92b53c932b65ceb44",
            "account": "0xa35EEbcac6c93ad12b4dedFB6E651c6eeB252541",
            "initData": "0x9cc7a4860f0b0926603c77f2e17ec5408745d45e2b668a287ed04e3aab0ea3d0",
            "salt": "0x7c1e3133c5e040bb7fc55cda56e3c1998a2e33373c0850e92b53c932b65ceb44"
        }
    }
]
```


As it is clear from the emitted data, the `AccountCreated` returns the hash of the `initData` while the `AccountCreated1` returns the original `bytes` variable.

### 9.2.5 Tools Used

Manual

### 9.2.6 Recommendations

```
    /// @notice Emitted when a new Smart Account is created, capturing the account details and
    ↪  associated module configurations.
-    event AccountCreated(address indexed account, bytes indexed initData, bytes32 indexed salt);
+    event AccountCreated(address indexed account, bytes initData, bytes32 indexed salt);
```

## 9.3 L-03. entryPoint() function cannot be overridden

*Submitted by h2134, Pelz. Selected submission by: h2134.*

### 9.3.1 Summary

entryPoint() function cannot be overridden, due to that it has no `virtual` keyword.

### 9.3.2 Vulnerability Details

As stated by the comment of `entryPoint()` function, this function is supposed to be overriden to return a different EntryPoint address if needed.

```
    /// @notice Retrieves the address of the EntryPoint contract, currently using version 0.7.
    /// @dev This function returns the address of the canonical ERC4337 EntryPoint contract.
@>  /// It can be overridden to return a different EntryPoint address if needed.
    /// @return The address of the EntryPoint contract.
    function entryPoint() external view returns (address) {
        return _ENTRYPOINT;
    }
```

However, this function is not defined by using `virtual` keyword, hence it cannot be overridden as expected.

### 9.3.3 Impact

`entryPoint()` function cannot be overridden.

### 9.3.4 Tools Used

Manual Review

### 9.3.5 Recommendations

Add `virtual` keyword to `entryPoint()` function so that it can be overridden

```
-   function entryPoint() external view returns (address) {
+   function entryPoint() external virtual view returns (address) {
        return _ENTRYPOINT;
    }
```

## 9.4 L-04. `Nexus.validateUserOp()` violates the EIP-4337 specification

*Submitted by zhuying, MiloTruck. Selected submission by: MiloTruck.*

### 9.4.1 Summary

`validateUserOp()` in nexus accounts do not revert when the validator specified is not installed, violating the EIP-4337 specification.

### 9.4.2 Vulnerability Details

According to EIP-4337, `validateUserOp()` must revert if it encounters any error apart from a signature mismatch (ie. `PackedUserOperation.signature` is not a valid signature of `userOpHash`):

> If the account does not support signature aggregation, it MUST validate the signature is a valid signature of the `userOpHash`, and SHOULD return SIG_VALIDATION_FAILED (and not revert) on signature mismatch. Any other error MUST revert.

However, if the validator specified in `PackedUserOperation.nonce` is not installed in the smart account, `Nexus.validateUserOp()` returns `SIG_VALIDATION_FAILED` instead of reverting:

Nexus.sol#L104-L105

```
// Check if validator is not enabled. If not, return VALIDATION_FAILED.
if (!_isValidatorInstalled(validator)) return VALIDATION_FAILED;
```

This is a violation of the EIP-4337 specification - `validator` not being installed is not a mismatch between `userOpHash` and the signature provided, so the function should revert.

### 9.4.3 Impact

Violation of the EIP-4337 specification could break composability with the `EntryPoint` contract and cause integration issues.

### 9.4.4 Recommendations

Instead of returning `VALIDATION_FAILED`, the function should revert:

```
- if (!_isValidatorInstalled(validator)) return VALIDATION_FAILED;
+ require(_isValidatorInstalled(validator), InvalidModule(validator));
```

## 9.5 L-05. Missing `_isInitialized(msg.sender)` check in `K1Validator.transferOwnership()`

*Submitted by MiloTruck.*

### 9.5.1 Summary

`K1Validator.transferOwnership()` does not check if the module is initialized for a smart account.

### 9.5.2 Vulnerability Details

`K1Validator.transferOwnership()` does not check if the module has been initialized for a smart account before setting `smartAccountOwners` to the new owner:

K1Validator.sol#L66-L71

```
function transferOwnership(address newOwner) external {
    require(newOwner != address(0), ZeroAddressNotAllowed());
    require(!_isContract(newOwner), NewOwnerIsContract());

    smartAccountOwners[msg.sender] = newOwner;
}
```

Therefore, it is possible for a smart account to call `transferOwnership()` while the `K1Validator` module has not been initialized for it. If this occurs, it will no longer be possible for the module to be installed for the smart account, since `onInstall()` will revert due to the `_isInitialized()` check:

K1Validator.sol#L53

```
function onInstall(bytes calldata data) external {
    require(data.length != 0, NoOwnerProvided());
    require(!_isInitialized(msg.sender), ModuleAlreadyInitialized());
```

K1Validator.sol#L133-L135

```
function _isInitialized(address smartAccount) private view returns (bool) {
    return smartAccountOwners[smartAccount] != address(0);
}
```

As such, a smart account could be permanently prevented from using a `K1Validator` module if it calls `transferOwnership()` before initialization.

A realistic scenario where this could occur:

- Smart account owner sends a transaction calling `transferOwnership()`.
- The transaction is not executed for a long period of time.
- Smart account owner sends a transaction calling `uninstallModule()` to uninstall the module:
  - `onUninstall()` is called, which resets the module to uninitialized.
- The transaction calling `transferOwnership()` is executed afterwards.
- Now, the smart account can no longer re-install the `K1Validator` module.

### 9.5.3 Impact

Smart accounts can be locked out of using the `K1Validator` module permanently.

### 9.5.4 Recommendations

In `transferOwnership()`, consider checking if the module is initialized for the calling smart account with `_isInitialized(msg.sender)`.

## 9.6 L-06. Fallback Handlers Can Be Installed with Invalid Calltype

*Submitted by DemoreXTess.*

### 9.6.1 Summary

In ModuleManager contract, _installFallbackHandler(module, initData) installs fallback handler without checking the calltype value. This fallback handler cannot be called by Nexus.

### 9.6.2 Vulnerability Details

_installFallbackHandler(module, initData) installs fallback handler with following lines:

```
function _installFallbackHandler(address handler, bytes calldata params) internal virtual
↪  withRegistry(handler, MODULE_TYPE_FALLBACK) {
        if (!IFallback(handler).isModuleType(MODULE_TYPE_FALLBACK)) revert
        ↪  MismatchModuleTypeId(MODULE_TYPE_FALLBACK);
        // Extract the function selector from the provided parameters.
        bytes4 selector = bytes4(params[0:4]);

        // Extract the call type from the provided parameters.
        CallType calltype = CallType.wrap(bytes1(params[4]));

        // Extract the initialization data from the provided parameters.
        bytes memory initData = params[5:];

        // Revert if the selector is either `onInstall(bytes)` (0x6d61fe70) or `onUninstall(bytes)`
        ↪  (0x8a91b0e3).
        // These selectors are explicitly forbidden to prevent security vulnerabilities.
        // Allowing these selectors would enable unauthorized users to uninstall and reinstall critical
        ↪  modules.
        // If a validator module is uninstalled and reinstalled without proper authorization, it can
        ↪  compromise
        // the account's security and integrity. By restricting these selectors, we ensure that the
        ↪  fallback handler
        // cannot be manipulated to disrupt the expected behavior and security of the account.
        require(!(selector == bytes4(0x6d61fe70) || selector == bytes4(0x8a91b0e3)),
        ↪  FallbackSelectorForbidden());

        // Revert if a fallback handler is already installed for the given selector.
        // This check ensures that we do not overwrite an existing fallback handler, which could lead to
        ↪  unexpected behavior.
        require(!_isFallbackHandlerInstalled(selector), FallbackAlreadyInstalledForSelector(selector));

        // Store the fallback handler and its call type in the account storage.
        // This maps the function selector to the specified fallback handler and call type.
        _getAccountStorage().fallbacks[selector] = FallbackHandler(handler, calltype);

        // Invoke the `onInstall` function of the fallback handler with the provided initialization
        ↪  data.
        // This step allows the fallback handler to perform any necessary setup or initialization.
        IFallback(handler).onInstall(initData);
    }
```

"Calltype is bytes1;" is user-defined type. This calltype variable is decoded from params variable. But there are at most 2 type of calltype defined in the Nexus:

- CALLTYPE_STATIC
- CALLTYPE_SINGLE

So, it can configure calltype wrong.

### 9.6.3 Impact

ModuleManager's fallback function defined as:

```
fallback() external payable override(Receiver) receiverFallback {
        FallbackHandler storage $fallbackHandler = _getAccountStorage().fallbacks[msg.sig];
        address handler = $fallbackHandler.handler;
        CallType calltype = $fallbackHandler.calltype;
        require(handler != address(0), MissingFallbackHandler(msg.sig));

        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, 0, 0, add(calldatasize(), 20), 0, 0)) {
                    returndatacopy(0, 0, returndatasize())
                    revert(0, returndatasize())
                }
                returndatacopy(0, 0, returndatasize())
                return(0, returndatasize())
            }
        }
    }
```

It's payable and open to wrong calltype vulnerability. It checks the calltype in here with 2 if statement and it will not catch any of those two if statement and transaction will and with no failure.

Scenario:

Bob installed a fallback handler to his Nexus with wrong calltype (example: 0x20) Later, Alice wants to call that fallback handler with 1 ether for execution Both if statements won't be catched because of invalid calltype and 1 ether will be on the Bob's Nexus without failure

### 9.6.4 Tools Used

Manual review

### 9.6.5 Recommendations

Following check will solve the problem

```
@@ -280,6 +281,7 @@ abstract contract ModuleManager is Storage, Receiver, EIP712, IModuleManagerEven
         // Extract the call type from the provided parameters.
         CallType calltype = CallType.wrap(bytes1(params[4]));

+        require(calltype == CALLTYPE_SINGLE || calltype == CALLTYPE_STATIC, InvalidCallType());
         // Extract the initialization data from the provided parameters.
         bytes memory initData = params[5:];
```