THREE SIGMA

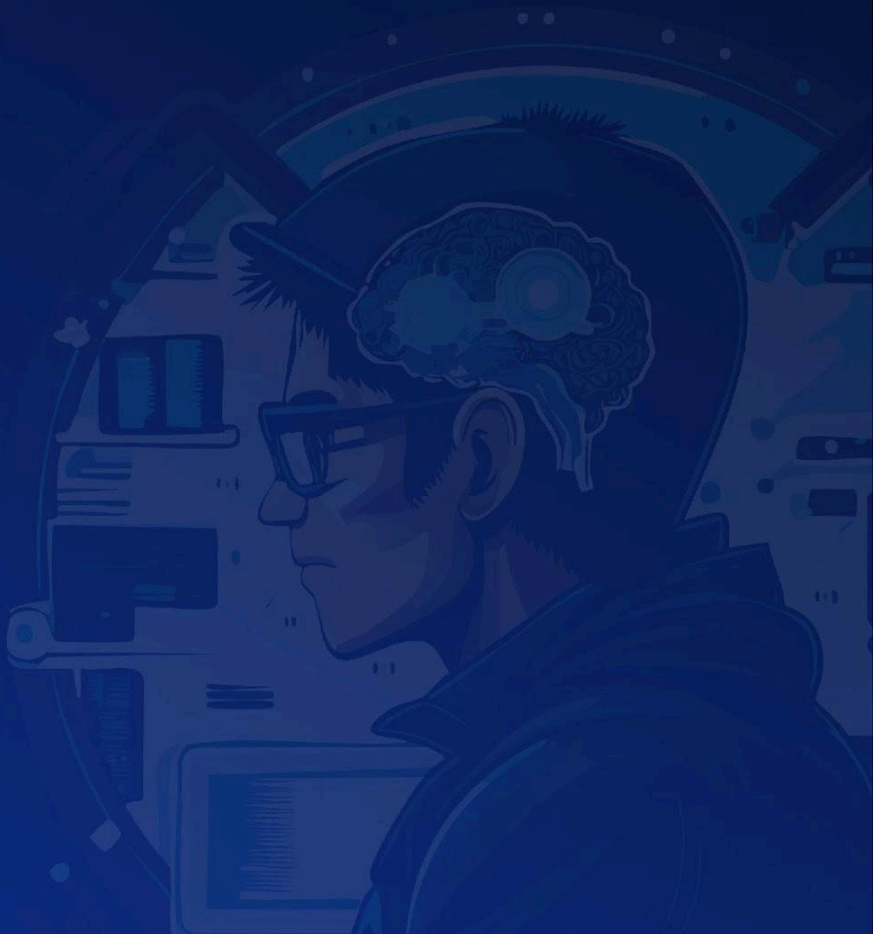
Felix EIP-7702-stock

Security Review

Disclaimer

Security Review

Felix EIP-7702-stock



Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

Felix EIP-7702-stock

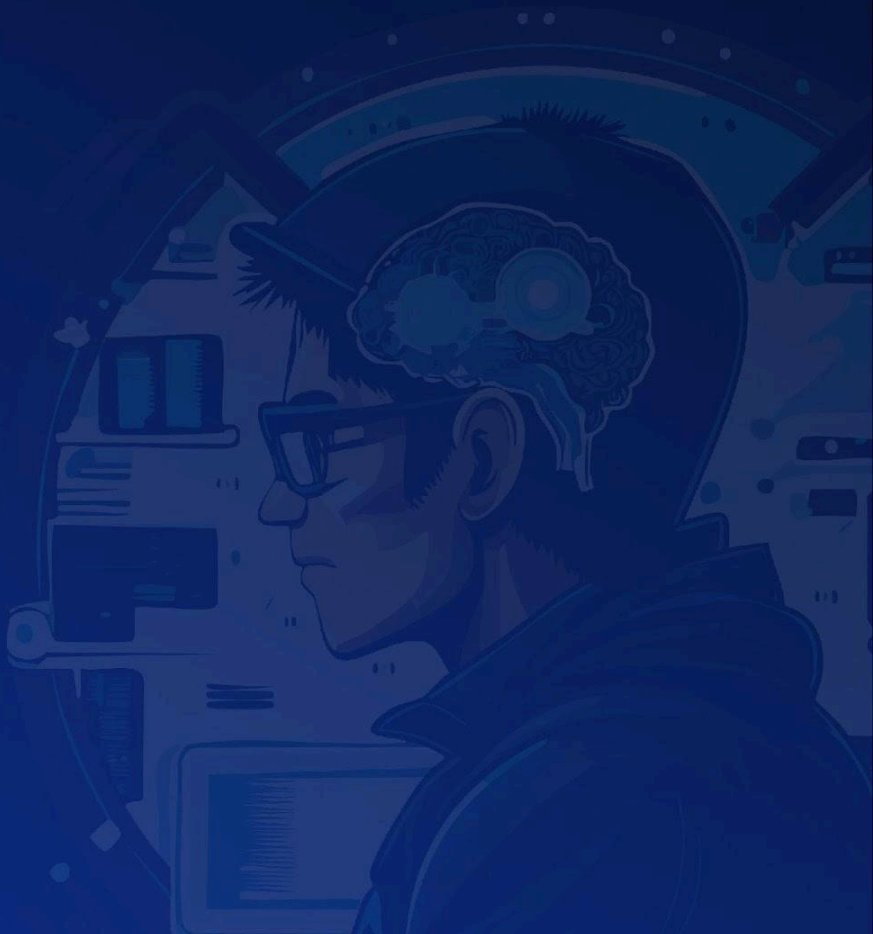
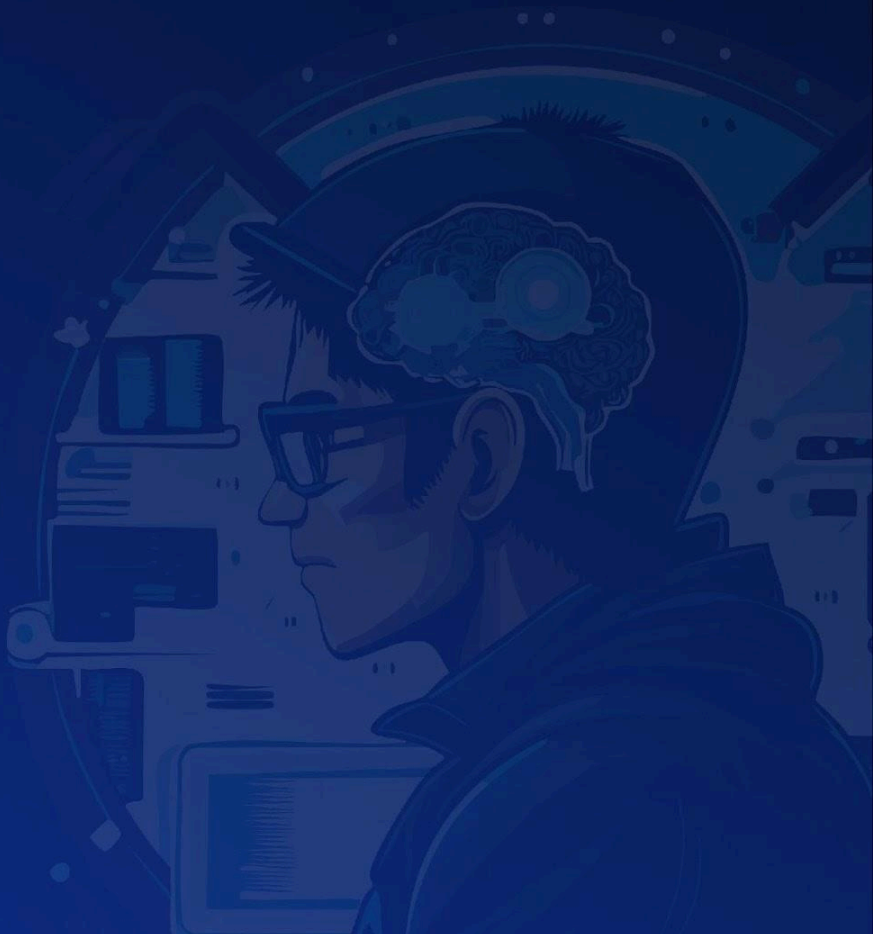


Table of Contents

Summary

Security Review

Felix EIP-7702-stock



Summary

Three Sigma audited Felix in a 1.6 person week engagement. The audit was conducted from 19/12/2025 to 22/12/2025.

Protocol Description

EIP-7702 Stock Trading System is a comprehensive smart contract system for executing batch transactions via EIP-7702 and facilitating tokenized asset trading through Ondo's GM Token Manager. This project enables gasless transactions through relayers and provides a brokerage service for minting and redeeming tokenized equities and commodities.

The project consists of three main components:

1. **Executor** - Enables EIP-7702 batch transaction execution with optional relayer support

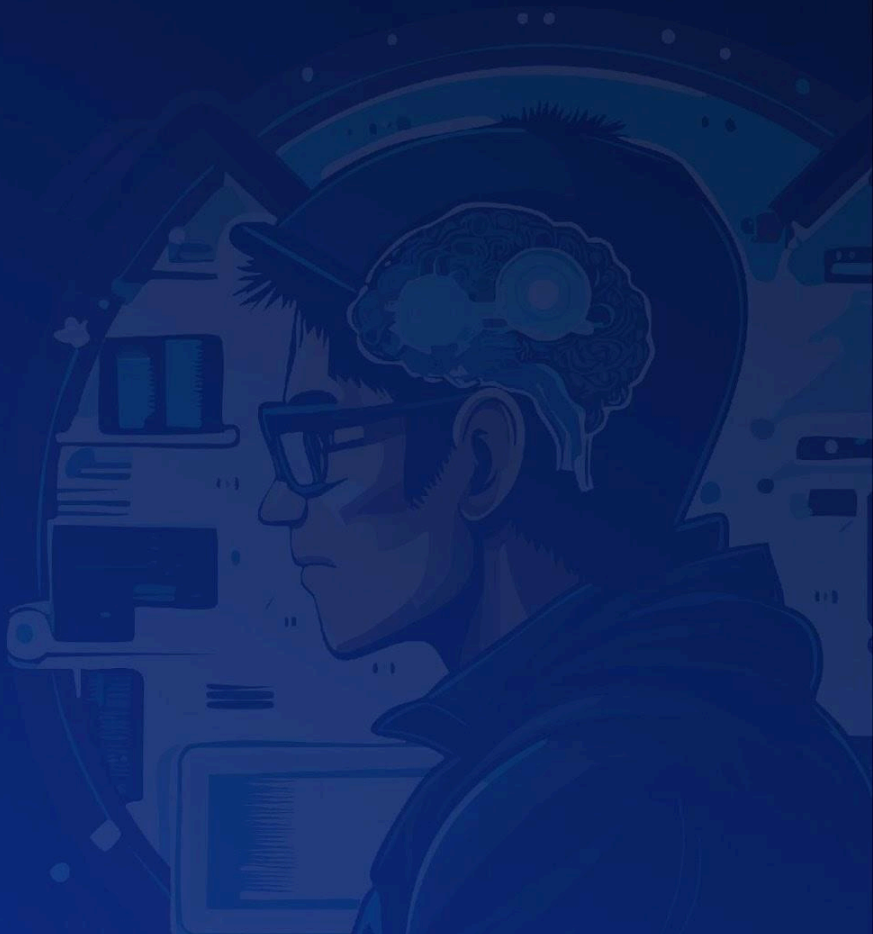
2. **RelayerGates** - Access control system for managing relayer permissions

3. **Retailer** - Brokerage contract for interacting with Ondo GM Token Manager to mint/redeem tokenized assets

Scope

Security Review

Felix EIP-7702-stock



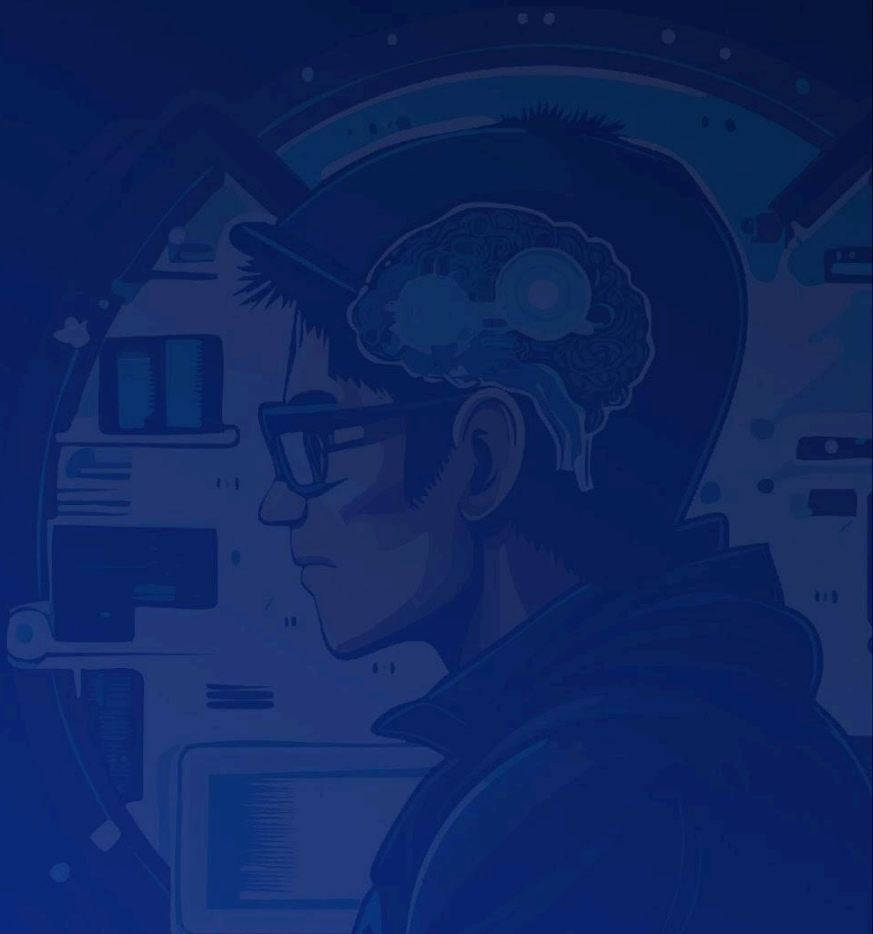
Scope

Filepath	nSLOC
src/Retailer.sol	85
src/RelayerGates.sol	61
src/Executor.sol	106
src/RetailerUpgradeable.sol	107
Total	**359**

Methodology

Security Review

Felix EIP-7702-stock



Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

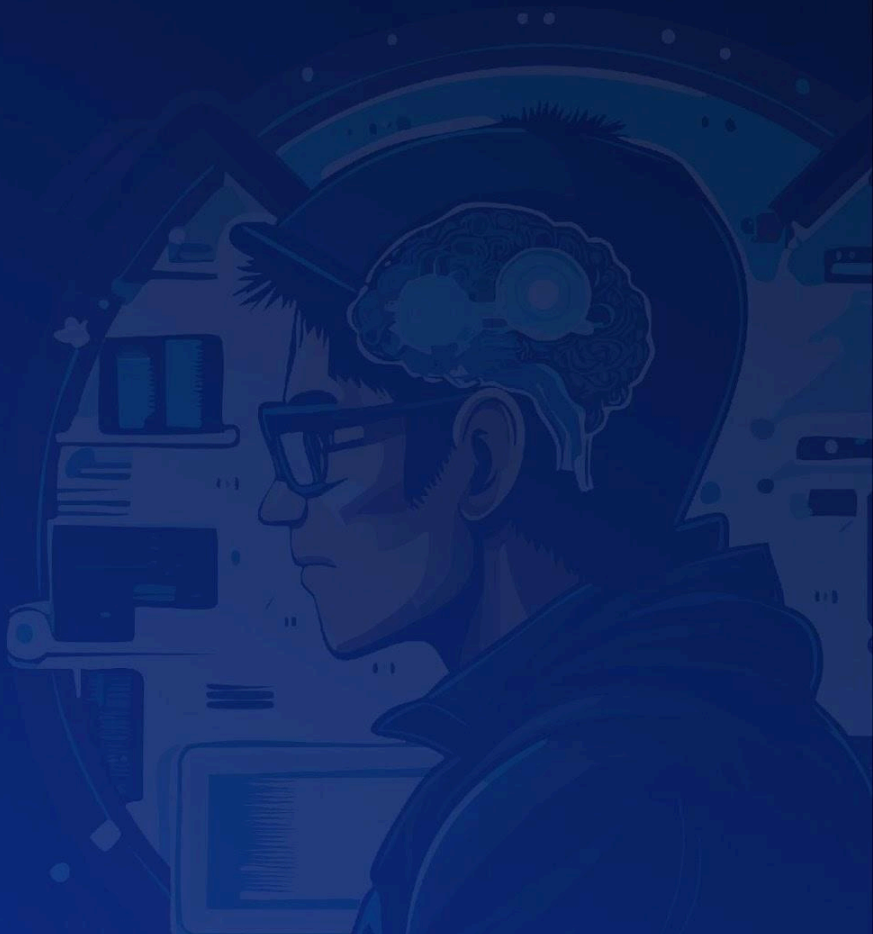
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Felix EIP-7702-stock



Project Dashboard

Application Summary

Name	Felix
Repository	https://github.com/felixprotocol/EIP-7702-stock
Commit	9ac946d
Language	Solidity
Platform	Hyperliquid

Engagement Summary

Timeline	19/12/2025 to 22/12/2025
№ of Auditors	2
Review Time	1.6 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	0	0	0
Medium	1	1	0
Low	4	4	0
None	3	2	1

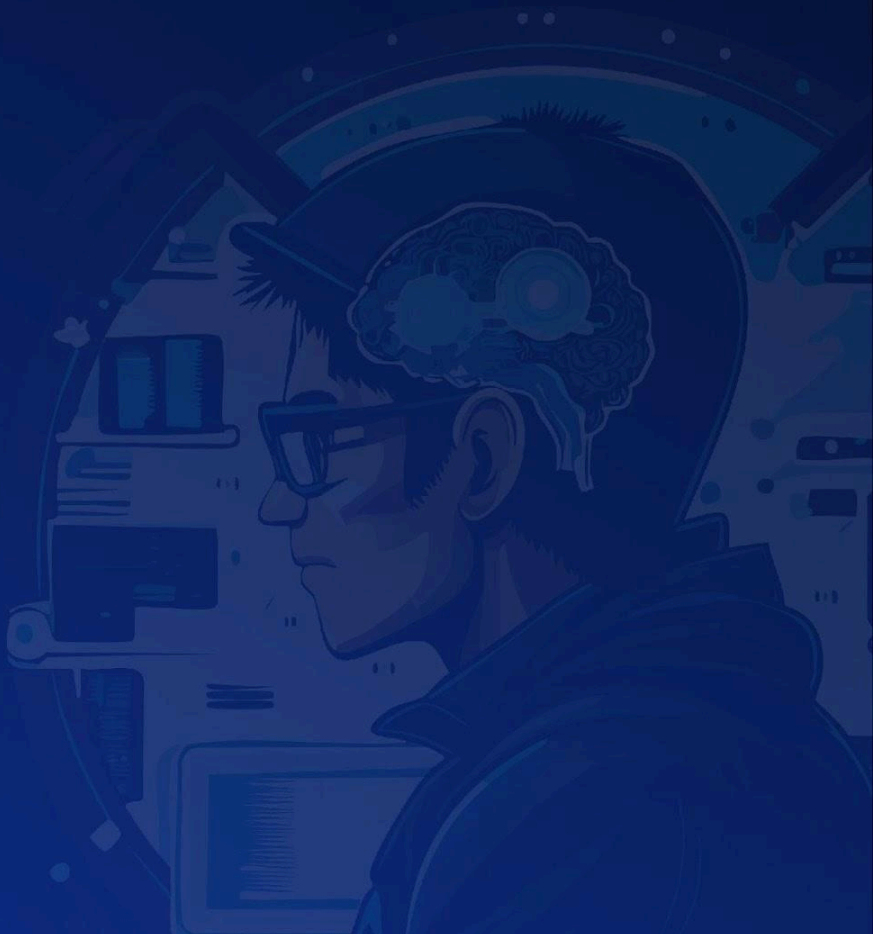
Category Breakdown

Suggestion	3
Documentation	0
Bug	6
Optimization	0
Good Code Practices	0

Risk Section

Security Review

Felix EIP-7702-stock



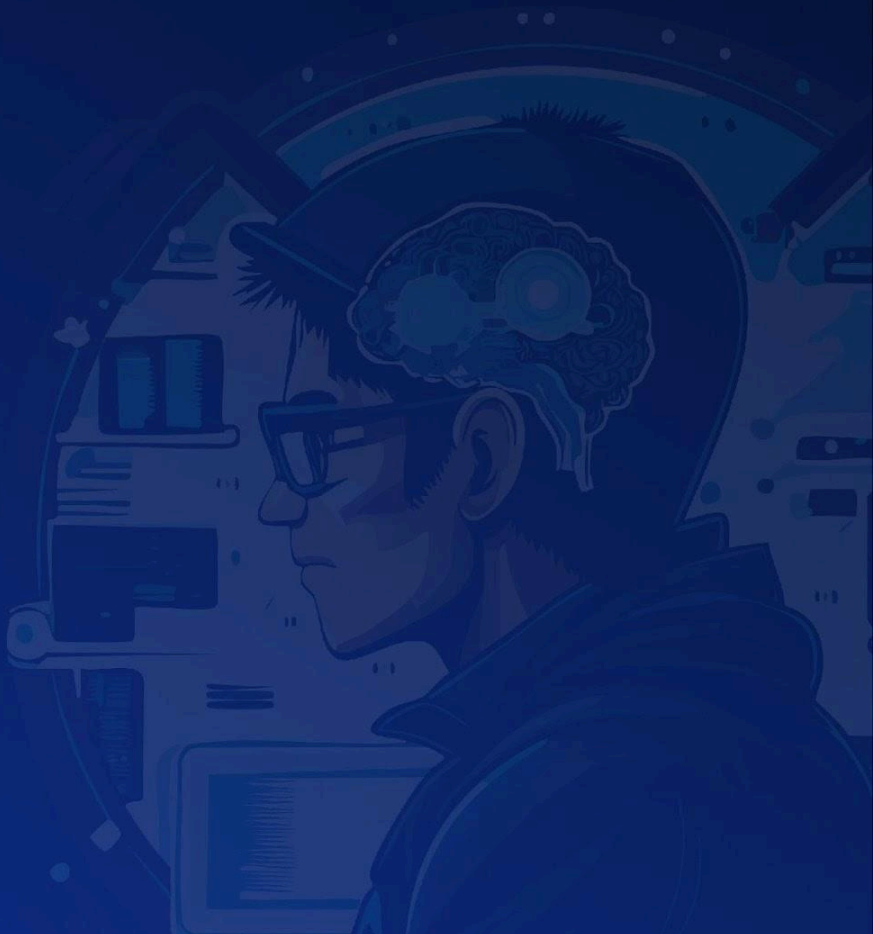
Risk Section

No risks were identified.

Findings

Security Review

Felix EIP-7702-stock



Findings

3S-Felix-C01

Missing **token** and **amount** fields in struct hash allows relayer to drain arbitrary funds

Id	3S-Felix-C01
Classification	Critical
Impact	Critical
Likelihood	Medium
Category	Bug
Status	Addressed in #1de8927.

Description

The **Executor** contract enables gasless transactions through a relayer mechanism. When users want to execute batched calls via a relayer, they sign a **BatchExecuteData** struct that includes the calls to execute, a deadline, nonce, and the **token** and **amount** they agree to pay as compensation to the relayer.

The **_BATCH_EXECUTE_DATA_TYPEHASH** correctly defines all fields that should be signed:

```
    bytes32 private constant _BATCH_EXECUTE_DATA_TYPEHASH =
        keccak256("BatchExecuteData(Call[] calls,uint256 deadline,uint256
nonce,address token,uint256 amount)");
```

However, both overloads of **Executor::_computeStructHash** omit the **token** and **amount** fields when computing the actual digest:

```
    function _computeStructHash(BatchExecuteData calldata _batchExecuteData)
private returns (bytes32) {
        return keccak256(
            abi.encode(
                _BATCH_EXECUTE_DATA_TYPEHASH,
                keccak256(abi.encode(_batchExecuteData.calls)),
```

```
            _batchExecuteData.deadline,
            _useNonce()
        )
    );
}
```

This mismatch means the user's signature does not bind them to any specific **token** or **amount** value. A malicious relayer can inject arbitrary values for these fields, and the signature verification will still pass.

Steps to Reproduce:

1. A user signs a **BatchExecuteData** containing only their intended **calls**, **deadline**, and **nonce** (since **token** and **amount** are not part of the digest)

2. The malicious relayer constructs a **BatchExecuteData** using the user's signed values but injects arbitrary **token** and **amount** values (e.g., a high-value token and the user's entire balance)

3. The relayer calls **Executor::batchExecuteRelayed** with this crafted **_batchExecuteData** and the user's valid signature

4. Signature verification passes because **token** and **amount** are not included in the digest computation

5. The contract executes the user's intended calls, then transfers the attacker-specified token and amount to the relayer via **_pushFundsToRelayer**

Recommendation

Include **token** and **amount** in the struct hash computation to ensure users explicitly authorize the payment terms:

```
 function _computeStructHash(BatchExecuteData calldata _batchExecuteData)
private returns (bytes32) {
    return keccak256(
        abi.encode(
            _BATCH_EXECUTE_DATA_TYPEHASH,
            keccak256(abi.encode(_batchExecuteData.calls)),
            _batchExecuteData.deadline,
-           _useNonce()
+           _useNonce(),
+           _batchExecuteData.token,
+           _batchExecuteData.amount
        )
```

```
    );
}
 function _computeStructHash(BatchExecuteData calldata _batchExecuteData,
uint256 _nonce)
    private
    pure
    returns (bytes32)
{
    return keccak256(
        abi.encode(
            _BATCH_EXECUTE_DATA_TYPEHASH,
            keccak256(abi.encode(_batchExecuteData.calls)),
            _batchExecuteData.deadline,
-           _nonce
+            _nonce,
+            _batchExecuteData.token,
+            _batchExecuteData.amount
        )
    );
}
```

3S-Felix-M01

Missing **receive/payable** fallback in **Executor** leads to reverted native transfers to EIP-7702 delegated EOAs

Id	3S-Felix-M01
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #fd1be48.

Description

Executor is intended to be delegated to an EOA via EIP-7702 so the EOA can execute batched calls through **Executor::batchExecute** and **Executor::batchExecuteRelayed**. After delegation, the EOA address behaves like a contract for call semantics, meaning native ETH transfers to the EOA are processed by the delegated runtime code. Since **Executor** does not implement a **receive()** function and does not provide a payable fallback, any incoming native transfer with empty calldata to a delegated EOA will revert, preventing the EOA from accepting plain ETH transfers while the delegation is active.

#

```
contract NativeTransferToDelegatedEOAPoC is BaseTest {
    function test_poc_native_transfer_to_delegated_eoa_reverts() public {
        // Fund this test contract so it can attempt native transfers.
        vm.deal(address(this), 1 ether);
        // Sanity: a plain EOA (no delegation) can receive native ETH fine.
        address plainEoa = makeAddr("plainEoa");
        payable(plainEoa).transfer(1 wei);
        assertEq(plainEoa.balance, 1 wei, "plain EOA should receive native");
        // PoC: `BOB` delegates to `Executor` in `BaseTest.setUp()`.
        // A plain ETH transfer (empty calldata) now executes `Executor` runtime, which
        // lacks a `receive()` function, so the transfer reverts.
        uint256 bobBalanceBefore = BOB.balance;
        vm.expectRevert();
        payable(BOB).transfer(1 wei);
        assertEq(BOB.balance, bobBalanceBefore, "BOB balance should be unchanged
on revert");
```

```
        // Same behavior observable via low-level call:
        (bool success,) = payable(BOB).call{value: 1 wei}("");
        assertFalse(success, "low-level native call should fail for delegated EOA without
receive()");
    }
}
```

Recommendation

Add a payable **receive** to **Executor** to accept plain native transfers, and optionally add a payable **fallback()** if you want to accept non-matching calldata without reverting.

```
 contract Executor is IExecutor, EIP712, Context, SanityChecks, Events {
+    receive() external payable {}
 }
```

3S-Felix-L01

Slippage protection in **redeemWithAttestation** does not account for brokerage fee

Id	3S-Felix-L01
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #7a5e101.

Description

The **Retailer::redeemWithAttestation** and **RetailerUpgradeable::redeemWithAttestation** functions allow users to redeem GM tokens for a receive token (e.g., USDC) through the Ondo GMTokenManager. These functions act as a brokerage layer that charges a fee on the received tokens before transferring the remaining balance to the user.

The **_minimumReceiveAmount** parameter is intended to provide slippage protection, ensuring users receive at least their specified minimum amount. However, the current implementation passes this value directly to **i_gmTokenManager.redeemWithAttestation**, which applies the slippage check before the brokerage fee is deducted.

This means users cannot set slippage protection that accounts for the brokerage fee. If a user sets **_minimumReceiveAmount** to 100 tokens and the brokerage fee is 1%, the GMTokenManager will ensure at least 100 tokens are received, but the user will only receive approximately 99 tokens after the fee deduction.

Recommendation

Move the slippage check to occur after the brokerage fee is deducted. Pass 0 to the GMTokenManager's **_minimumReceiveAmount** parameter and perform the validation within the Retailer contract.

For **Retailer.sol**:

```
function redeemWithAttestation(
    IGMTokenManager.Quote calldata _quote,
```

```
    bytes memory _signature,
    address _receiveToken,
    uint256 _minimumReceiveAmount
) external {
    _requireNotZeroAddress(_receiveToken);
    _requireNotZeroAmount(_minimumReceiveAmount);
    _requireValidDepositToken(_receiveToken, i_depositToken);
    _pullToken(_quote.asset, _quote.quantity);
    if (!s_approvedAssets[_quote.asset]) {
        s_approvedAssets[_quote.asset] = true;
        _approveGMTokenManager(_quote.asset);
    }
-   i_gmTokenManager.redeemWithAttestation(_quote, _signature, _receiveToken,
_minimumReceiveAmount);
+    i_gmTokenManager.redeemWithAttestation(_quote, _signature, _receiveToken, 0);
    uint256 receiveTokenBalance = _getBalance(_receiveToken);
    uint256 brokerageFee = s_brokerageFee;
    if (brokerageFee > 0) {
        uint256 feeAmount = _calculateBrokerageFee(receiveTokenBalance,
brokerageFee);
        receiveTokenBalance -= feeAmount;
        _pushToken(_receiveToken, feeAmount);
    }
+   require(receiveTokenBalance >= _minimumReceiveAmount, "Slippage
exceeded");
    IERC20(_receiveToken).safeTransfer(_msgSender(), receiveTokenBalance);
    _emitRedeemedWithAttestation(_quote, _receiveToken,
_minimumReceiveAmount, _msgSender());
}
```

For **RetailerUpgradeable.sol**:

```
function redeemWithAttestation(
    IGMTokenManager.Quote calldata _quote,
    bytes memory _signature,
    address _receiveToken,
    uint256 _minimumReceiveAmount
) external {
    _requireNotZeroAddress(_receiveToken);
    _requireNotZeroAmount(_minimumReceiveAmount);
    _requireValidDepositToken(_receiveToken, i_depositToken);
    _pullToken(_quote.asset, _quote.quantity);
    if (!_getApprovedAssets(_quote.asset)) {
```

```
        _setApprovedAssets(_quote.asset, true);
        _approveGMTokenManager(_quote.asset);
    }
-   i_gmTokenManager.redeemWithAttestation(_quote, _signature, _receiveToken,
_minimumReceiveAmount);
+    i_gmTokenManager.redeemWithAttestation(_quote, _signature, _receiveToken, 0);
    uint256 receiveTokenBalance = _getBalance(_receiveToken);
    uint256 brokerageFee = _getBrokerageFee();
    if (brokerageFee > 0) {
        uint256 feeAmount = _calculateBrokerageFee(receiveTokenBalance,
brokerageFee);
        receiveTokenBalance -= feeAmount;
        _pushToken(_receiveToken, feeAmount);
    }
+    require(receiveTokenBalance >= _minimumReceiveAmount, "Slippage
exceeded");
    IERC20(_receiveToken).safeTransfer(_msgSender(), receiveTokenBalance);
    _emitRedeemedWithAttestation(_quote, _receiveToken, receiveTokenBalance,
_msgSender());
 }
```

3S-Felix-L02

Incorrect EIP-712 encoding of **Call** array in **Executor::_computeStructHash**

Id	3S-Felix-L02
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #198bb65.

Description

The **Executor** contract enables gasless batch transactions using EIP-7702. Users sign a **BatchExecuteData** struct off-chain, and relayers submit these signed transactions on their behalf via **Executor::batchExecuteRelayed**. The **_computeStructHash** function computes the EIP-712 struct hash of the **BatchExecuteData**, which is then used to recover and verify the signer's address.

The current implementation incorrectly encodes the **calls** array:

```
    /**
     * @notice Computes the struct hash of the batch execute data.
     * @param _batchExecuteData The batch execute data.
     * @return The struct hash of the batch execute data.
     */
    function _computeStructHash(BatchExecuteData calldata _batchExecuteData)
private returns (bytes32) {
        return keccak256(
            abi.encode(
                _BATCH_EXECUTE_DATA_TYPEHASH,
                keccak256(abi.encode(_batchExecuteData.calls)),
                _batchExecuteData.deadline,
                _useNonce()
            )
        );
    }
```

According to the EIP-712 specification, array values containing structs must be encoded as:

> "The array values are encoded as the keccak256 hash of the concatenated encodeData of their contents [...] The struct values are encoded recursively as hashStruct(value)."

This means each **Call** element must first be encoded using **hashStruct** (i.e., **keccak256(typeHash, encodedFields)**), then all resulting hashes must be concatenated and hashed. The current implementation uses **keccak256(abi.encode(_batchExecuteData.calls))**, which does not apply **hashStruct** to each element and does not hash the dynamic **callData** bytes field within each **Call** struct.

The same issue exists in the overloaded **_computeStructHash(BatchExecuteData, uint256)** function used by **getHashTypedDataV4**.

Recommendation

Add a **_CALL_TYPEHASH** constant and a helper function to correctly encode the **Call** array per EIP-712. Update both **_computeStructHash** overloads to use the new encoding:

```
+ /// @dev Typehash for the Call struct.
+ bytes32 private constant _CALL_TYPEHASH = keccak256("Call(address
target,uint256 value,bytes callData)");
+ /**
+  * @notice Computes the EIP-712 array encoding for a Call array.
+  * @param _calls The calls to encode.
+  * @return The keccak256 hash of the concatenated hashStruct values.
+  */
+ function _hashCallArray(Call[] calldata _calls) private pure returns (bytes32) {
+     bytes32[] memory hashes = new bytes32[](_calls.length);
+     for (uint256 i = 0; i < _calls.length; ++i) {
+         hashes[i] = keccak256(
+             abi.encode(
+                 _CALL_TYPEHASH,
+                 _calls[i].target,
+                 _calls[i].value,
+                 keccak256(_calls[i].callData)
+             )
+         );
+     }
+     return keccak256(abi.encodePacked(hashes));
+ }
  function _computeStructHash(BatchExecuteData calldata _batchExecuteData)
private returns (bytes32) {
      return keccak256(
```

```
        abi.encode(
            _BATCH_EXECUTE_DATA_TYPEHASH,
-           keccak256(abi.encode(_batchExecuteData.calls)),
+            _hashCallArray(_batchExecuteData.calls),
            _batchExecuteData.deadline,
            _useNonce()
        )
    );
 }
 function _computeStructHash(BatchExecuteData calldata _batchExecuteData,
uint256 _nonce)
    private
    pure
    returns (bytes32)
 {
    return keccak256(
        abi.encode(
            _BATCH_EXECUTE_DATA_TYPEHASH,
-           keccak256(abi.encode(_batchExecuteData.calls)),
+            _hashCallArray(_batchExecuteData.calls),
            _batchExecuteData.deadline,
            _nonce
        )
    );
 }
```

3S-Felix-L03

Incorrect EIP-712 typehash construction for **BatchExecuteData** omits referenced **Call** struct definition

Id	3S-Felix-L03
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #85e761c.

Description

The **Executor** contract implements EIP-712 typed structured data hashing to enable gasless relayed transactions via **Executor::batchExecuteRelayed**. This function allows a relayer to submit signed batch execution requests on behalf of users, where the signature is verified against an EIP-712 digest constructed using **_BATCH_EXECUTE_DATA_TYPEHASH**.

The **BatchExecuteData** struct contains a nested **Call[]** array, which references the **Call** struct type. According to the EIP-712 specification, when a struct type references other struct types, the referenced struct type definitions must be collected, sorted alphabetically by name, and appended to the primary type encoding.

The current implementation defines **_BATCH_EXECUTE_DATA_TYPEHASH** as:

keccak256("BatchExecuteData(Call[] calls,uint256 deadline,uint256 nonce,address token,uint256 amount)")

This is non-compliant with EIP-712 because the **Call** struct definition is not appended. The correct encoding should include the **Call** type definition after the **BatchExecuteData** definition.

This results in signature incompatibility with any external system (wallets, dApps, SDKs) that correctly implements EIP-712 signing. Users attempting to sign **BatchExecuteData** using a compliant EIP-712 implementation will produce signatures that cannot be verified by the contract, and vice versa.

Recommendation

Append the **Call** struct type definition to **_BATCH_EXECUTE_DATA_TYPEHASH** in accordance with EIP-712:

```
  bytes32 private constant _BATCH_EXECUTE_DATA_TYPEHASH =
-     keccak256("BatchExecuteData(Call[] calls,uint256 deadline,uint256
nonce,address token,uint256 amount)");
+     keccak256("BatchExecuteData(Call[] calls,uint256 deadline,uint256
nonce,address token,uint256 amount)Call(address target,uint256 value,bytes
callData)");
```

3S-Felix-L04

Incorrect parameter used in **RedeemedWithAttestation** event emission leads to inaccurate redeemed amount reporting

Id	3S-Felix-L04
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #c28e108.

Description

Retailer::redeemWithAttestation is intended to redeem a GM token position via the Ondo **IGMTokenManager**, apply the configured brokerage fee, and transfer the net **receiveToken** amount to **msg.sender**. After calling **IGMTokenManager::redeemWithAttestation**, the function computes **receiveTokenBalance**, subtracts the brokerage fee, and transfers the resulting net amount to **msg.sender**. However, the emitted **RedeemedWithAttestation** event reports **_minimumReceiveAmount** rather than the actual net amount transferred. Since **_minimumReceiveAmount** is an input constraint rather than the realized output, off-chain indexers, accounting systems, and monitoring that rely on **RedeemedWithAttestation** will record an incorrect redeemed amount. **RetailerUpgradeable::redeemWithAttestation** correctly emits the net **receiveTokenBalance**, creating an inconsistency between the two implementations.

Recommendation

Update **Retailer::redeemWithAttestation** to emit the post-fee **receiveTokenBalance** (the same value transferred to **msg.sender**) instead of **_minimumReceiveAmount**.

```
- _emitRedeemedWithAttestation(_quote, _receiveToken, _minimumReceiveAmount, _msgSender());
+ _emitRedeemedWithAttestation(_quote, _receiveToken, receiveTokenBalance, _msgSender());
```

3S-Felix-N01

Zero amount check in **Executor::batchExecuteRelayed** prevents relayer-sponsored transactions

Id	3S-Felix-N01
Classification	None
Category	Suggestion
Status	Addressed in .

Description

The **Executor::batchExecuteRelayed** function enables a designated relayer to execute a batch of calls on behalf of an EOA that has delegated to this Executor contract. The function validates the provided batch data, verifies the EOA's signature, executes all calls in the batch, and transfers a specified amount of tokens to the relayer as compensation for gas costs.

Currently, the function enforces a non-zero amount requirement via **_requireNotZeroAmount(_batchExecuteData.amount)**. This check prevents legitimate use cases where the relayer may choose to sponsor the transaction by waiving compensation. Relayer-sponsored transactions are a common pattern in meta-transaction systems to improve user experience, onboard new users, or as part of promotional activities.

Recommendation

Remove the **_requireNotZeroAmount** check to allow relayers the flexibility to sponsor transactions when **amount** is zero. The **_pushFundsToRelayer** function should handle zero amounts gracefully by either skipping the transfer or being a no-op.

```
 function batchExecuteRelayed(BatchExecuteData calldata _batchExecuteData,
Signature calldata _signature)
    external
    payable
    onlyRelayer
{
    _requireDeadlineNotExpired(_batchExecuteData.deadline);
    _requireValidToken(_batchExecuteData.token);
-   _requireNotZeroAmount(_batchExecuteData.amount);
```

```
    bytes32 _digest = _hashTypedDataV4(_computeStructHash(_batchExecuteData));
    address _signer = _computeSigner(_digest, _signature);
    _requireValidSigner(_signer);
    for (uint256 i = 0; i < _batchExecuteData.calls.length; ++i) {
        _execute(_batchExecuteData.calls[i]);
    }
    _pushFundsToRelayer(_batchExecuteData.token, _batchExecuteData.amount);
    _emitBatchExecuted(_msgSender(), _batchExecuteData.calls);
}
```

3S-Felix-N02

Attestation front-running allows attackers to steal user quotes in retailer contracts

Id	3S-Felix-N02
Classification	None
Category	Suggestion
Status	Acknowledged

Description

The **Retailer** and **RetailerUpgradeable** contracts serve as brokerage intermediaries between users and Ondo's **GMTokenManager**. They facilitate minting and redeeming GM tokens by forwarding user requests along with attestations (quotes) obtained from Ondo. Each attestation contains a unique **attestationId** that can only be used once, as enforced by **GMTokenManager::_verifyQuote**. The **_quote.userId** field identifies the retailer contract itself, not the individual user interacting with it.

The vulnerability exists in both **Retailer::mintWithAttestation** and **Retailer::redeemWithAttestation** (and their upgradeable counterparts). When a user calls these functions, they submit:

1. A **_quote** containing the attestation details

2. A **_signature** validating the quote

3. Token transfer parameters

The contract pulls tokens from **msg.sender**, deducts the brokerage fee, and forwards the request to **GMTokenManager**. However, there is no mechanism binding the attestation to the specific user submitting the transaction. The attestation is tied only to the retailer contract's identity, meaning any caller can use it.

Since attestation data and signatures are submitted as calldata, they are visible in the mempool before transaction inclusion. An attacker monitoring the mempool can extract this data and submit their own transaction with higher gas, effectively stealing the attestation.

Note: The protocol is currently deployed on HyperEVM, which does not have a public mempool. As a result, this front-running attack vector is not exploitable in the current deployment. However, this issue would become exploitable if the protocol is deployed on any chain with a public mempool.

Steps to reproduce:

1. Alice requests an attestation from Ondo through the retailer service for minting GM tokens

2. The retailer service receives a quote with **userId** set to the retailer contract's ID

3. Alice receives the quote and signature, then submits a transaction calling **Retailer::mintWithAttestation** with her deposit tokens

4. An attacker (Bob) monitors the mempool and observes Alice's pending transaction

5. Bob extracts **_quote** and **_signature** from Alice's transaction calldata

6. Bob submits an identical call to **mintWithAttestation** with a higher gas price, using his own deposit tokens

7. Bob's transaction is included first, successfully consuming the attestation and receiving the GM tokens

8. Alice's transaction reverts because the **attestationId** has already been used

The same attack vector applies to **redeemWithAttestation**, where an attacker can steal redemption quotes.

Recommendation

Introduce a dedicated signer role and use EIP-712 typed data signatures to bind each attestation to a specific user address.

```
+ import {EIP712} from "@openzeppelin/contracts/utils/cryptography/EIP712.sol";
+ import {ECDSA} from "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
- contract Retailer is SanityChecks, Context, Ownable2Step, Events {
+ contract Retailer is SanityChecks, Context, Ownable2Step, Events, EIP712 {
      using SafeERC20 for IERC20;
+     using ECDSA for bytes32;
+     bytes32 private constant _USER_BINDING_TYPEHASH =
+         keccak256("UserBinding(uint256 attestationId,address user)");
      IGMTokenManager public immutable i_gmTokenManager;
      address public immutable i_depositToken;
      uint256 public s_brokerageFee;
      mapping(address asset => bool isApproved) public s_approvedAssets;
+     address public s_attestationSigner;
-     constructor(address _gmTokenManager, address _depositToken, address
_owner, uint256 _brokerageFee)
-         Ownable(_owner)
+     constructor(address _gmTokenManager, address _depositToken, address
_owner, uint256 _brokerageFee, address _attestationSigner)
+         Ownable(_owner)
+         EIP712("FLX Retailer", "1")
```

```
    {
        // ... existing checks ...
+       _requireNotZeroAddress(_attestationSigner);
+       s_attestationSigner = _attestationSigner;
    }
+   function setAttestationSigner(address _signer) external onlyOwner {
+       _requireNotZeroAddress(_signer);
+       s_attestationSigner = _signer;
+   }
+   function _verifyUserBinding(
+       uint256 _attestationId,
+       address _user,
+       bytes memory _userBindingSignature
+   ) internal view {
+       bytes32 structHash = keccak256(abi.encode(_USER_BINDING_TYPEHASH,
_attestationId, _user));
+       bytes32 digest = _hashTypedDataV4(structHash);
+       address recoveredSigner = digest.recover(_userBindingSignature);
+       require(recoveredSigner == s_attestationSigner, "Invalid user binding
signature");
+   }
    function mintWithAttestation(
        IGMTokenManager.Quote calldata _quote,
        bytes memory _signature,
        address _depositToken,
-       uint256 _depositTokenAmount
+       uint256 _depositTokenAmount,
+       bytes memory _userBindingSignature
    ) external {
+       _verifyUserBinding(_quote.attestationId, _msgSender(),
_userBindingSignature);
        // ... rest of function
    }
    function redeemWithAttestation(
        IGMTokenManager.Quote calldata _quote,
        bytes memory _signature,
        address _receiveToken,
-       uint256 _minimumReceiveAmount
+       uint256 _minimumReceiveAmount,
+       bytes memory _userBindingSignature
    ) external {
+       _verifyUserBinding(_quote.attestationId, _msgSender(),
_userBindingSignature);
        // ... rest of function
```

```
    }
  }
```

The retailer backend service must sign the EIP-712 typed data **UserBinding(attestationId, userAddress)** when issuing attestations, ensuring only the intended recipient can use each quote.

3S-Felix-N03

Using the unchained initializer for ownership setup allows missed parent initialization on future upgrades

Id	3S-Felix-N03
Classification	None
Category	Suggestion
Status	Addressed in #54f34fa.

Description

RetailerUpgradeable::__RetailerUpgradable_init is the internal initializer that configures the upgradeable retailer after proxy deployment by setting the contract owner, persisting the brokerage fee, and approving the GM token manager. It currently calls **Ownable2StepUpgradeable::__Ownable2Step_init_unchained** instead of **Ownable2StepUpgradeable::__Ownable2Step_init**. In OpenZeppelin's upgradeable pattern, **__X_init** is the canonical entry point intended to be called by integrators, while **__X_init_unchained** is an internal hook used when explicitly managing initializer ordering across complex inheritance. In the current OpenZeppelin version used here, both **__Ownable2Step_init** and **__Ownable2Step_init_unchained** are empty, so there is no immediate functional impact; however, relying on the unchained variant can become brittle if future dependency versions introduce initialization logic into **__Ownable2Step_init**, potentially causing that logic to be skipped.

Recommendation

Update **RetailerUpgradeable::__RetailerUpgradable_init** to call **__Ownable2Step_init()** instead of **__Ownable2Step_init_unchained()** to follow the intended OpenZeppelin initializer flow and reduce the risk of missed initialization if upstream logic changes.

```
-       __Ownable2Step_init_unchained();
+       __Ownable2Step_init();
```