Superform v2 contracts
Security Review

Cantina Managed review by:

MiloTruck, Lead Security Researcher
Ladboy233, Security Researcher

April 19, 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

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it 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 that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
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 table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (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 Mar 26th to Apr 5th the Cantina team conducted a review of superform-v2-contracts on commit hash 01e73378. The team identified a total of **38** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	3	3	0
Medium Risk	9	9	0
Low Risk	18	16	2
Gas Optimizations	0	0	0
Informational	8	7	1
Total	**38**	**35**	**3**

3 Findings

3.1 High Risk

3.1.1 Incomplete merkle leaf generation because of missing `PackedUserOperation` fields in `SuperMerkleValidator.sol`

Severity: High Risk

Context: SuperMerkleValidator.sol#L77-L96

Description: The code extract the `sender` and `nonce` and `callData` and `initCode` and `gasCodes` from the `PackedUserOperation` and generate the leaf for merkle root.

The full PackedUserOperation struct contains other field.

```
struct PackedUserOperation {
    address sender;
    uint256 nonce;
    bytes initCode;
    bytes callData;
    bytes32 accountGasLimits;
    uint256 preVerificationGas;
    bytes32 gasFees;
    bytes paymasterAndData;
    bytes signature;
}
```

The code ignores the fields below.

- `accountGasLimits`.
- `preVerificationGas`.
- `paymasterAndData`.
- `signature`.

Consider the example below.

1. Signer signs a ERC4337 operation and attach `paymasterAndData` and expect the paymaster cover the transaction fee.
2. However, the submitter replace `paymasterAndData` field with `0x empty byte` and execute the transaction.
3. The signer has to pay for transaction himself.

Recommendation: According to EIP 4337:

> Smart Contract Account Interface. The `userOpHash` is a hash over the `userOp` (except `signature`), `entryPoint` and `chainId`. The Smart Contract Account MUST validate that the signature is a valid signature of the `userOpHash`, and SHOULD return `SIG_VALIDATION_FAILED` (1) without reverting on signature mismatch. Any other error MUST revert.

The `userOpHash` and every `PackedUserOp` field should be used to generate the merkle leaf.

Superform: Fixed in PR 313.

Cantina Managed: Fix verified.

3.1.2 `isValidSignatureWithSender()` in validator modules fail to validate the ERC 1271 `hash`

Severity: High Risk

Context: SuperMerkleValidator.sol#L109-L131, SuperDestinationValidator.sol#L53-L73.

Description: In `SuperMerkleValidator` and `SuperDestinationValidator`, `isValidSignatureWithSender()` is not implemented according to the ERC 1271 and ERC 7579 specifications. More specifically, the function does not validate that `hash` was authorized by the account's owner. The ERC 1271 signature validation flow with an account as the owner is as follows:

- A DeFi contract calls ERC 1271's `isValidSignature()` to verify a `hash` is signed by an account.

- The account calls ERC 7579 validator's `isValidSignatureWithSender()` to verify the `hash` through its validator module. The purpose of this flow is to check the `hash` has been "authorized" by the account's owner, usually through a signature.

However, in the current implementation, the `hash` (which is the second parameter of `isValidSignatureWithSender()`) is not checked. Instead, the function extracts the merkle leaf and root from `data` (which is the signature data) and verifies the merkle root has been signed by the user.

This effectively means `isValidSignatureWithSender()` does not perform any validation, since anyone can pass a previously used merkle leaf and root in `data` and all checks will pass. As a result, anyone can call any function in a DeFi contract on behalf of an account with `SuperMerkleValidator` or `SuperDestinationValidator` installed, as long as the function uses ERC 1271 validation.

An example of a DeFi contract which uses ERC 1271 validation is OpenSea.

Additionally, the `sender` parameter, which is passed to `isInitialized`, is not the account address. Instead, it is the caller of `isValidSignature()`, which would be the DeFi contract in the example above.

Recommendation: Consider re-implementing `isValidSignatureWithSender()` to check that `hash` has been authorized by the account's owner, using the provided `signature`. An reference implementation of `isValidSignatureWIthSender()` in validator modules can be found in Biconomy's `K1Validator`.

Additionally, in `SuperMerkleValidator`, pass `msg.sender` to `_initialized` instead of `sender` as the account calls `isValidSignatureWithSender()`.

Superform: Fixed in PR 340 and PR 358.

Cantina Managed: Verified, the issue has been addressed by checking that `hash` is part of a merkle root signed by the account's owner.

3.1.3 `SuperLedger` accounting is incompatible with Gearbox and Fluid hooks

Severity: High Risk

Context: GearboxUnstakeHook.sol, FluidUnstakeHook.sol.

Description: The `SuperExecutor` and `SuperLedger` flow has to be revisited for non-vault integrations, namely the Gearbox and Fluid hooks. Currently, `SuperLedger` accounting only works for contracts that have the notion of shares and assets. In particular, `getPricePerShare()` fetches the share price and `SuperLedger` calculates profit from the average share price. However, Gearbox and Fluid staking do not have assets/shares and only transfers rewards to the user when a separate claim function is called. This does not work with the current `SuperExecutor` flow, since:

1. When staking through an `INFLOW` hook, an average share price is calculated based on rewards, which is incorrect.
2. When unstaking through an `OUTFLOW` hook, only the staking token is transferred to the user. As such, it is not possible to charge a fee on rewards when the `OUTFLOW` hook is called, which the current `SuperExecutor` flow does.

(2) is particularly problematic since the current implementation charges a fee on staking tokens.

For example, in Fluid staking, the staking token is fUSDT and the reward token is FLUID. As such, fees should be charged on FLUID when `FluidClaimRewardHook` calls `IFluidLendingStakingRewards.getReward()`, but instead, fees are charged on fUSDT when `FluidUnstakeHook` calls `IFluidLendingStakingRewards.withdraw()`. Additionally, `usedShares` is not set in `preExecute()` for `GearboxUnstakeHook` and `FluidUnstakeHook`. This causes `costBasis` to become `0` in `SuperLedger` and the entire withdrawn amount is taken as profit.

Recommendation: A possible solution would be to have the Fluid/Gearbox staking and unstaking hooks as `NONACCOUNTING`, and the claim hooks as `OUTFLOW`. To charge fees on rewards, create a new ledger contract which applies a flat percentage fee on the amount of rewards received (i.e. `outAmount`).

Superform: Fixed in PR 345 and PR 365.

Cantina Managed: Verified, the relevant staking/unstaking hooks have been changed to `NONACCOUNTING` and all claim hooks are now `OUTFLOW`. A new `FlatFeeLedger` has been implemented to charge a percentage fee on rewards.

3.2 Medium Risk

3.2.1 Lack of `yieldSource` input validation tampers `SuperLedger` shares and accumulator cost accounting

Severity: Medium Risk

Context: SuperExecutor.sol#L110-L125

Description: The user can supply any `yieldSource` and then trigger `ISuperLedger(config.ledger).updateAccounting`.

Then the code take a snapshot of user's share and accumulator cost when triggering INFLOW hook action.

```
uint256 pps = IYieldSourceOracle(config.yieldSourceOracle).getPricePerShare(yieldSource);
if (pps == 0) revert INVALID_PRICE();

if (isInflow) {
    _takeSnapshot(
        user, amountSharesOrAssets, pps, IYieldSourceOracle(config.yieldSourceOracle).decimals(yieldSource)
    );

    emit AccountingInflow(user, config.yieldSourceOracle, yieldSource, amountSharesOrAssets, pps);
    return 0;
}
```

The oracle still have to consult the yield source data to get the `price per share` data:

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

Then a malicious yield source can tamper and inflate the `SuperLedger` shares and accumulator cost accounting by returning inflated `getPricePerShare`.

```
function _takeSnapshot(address user, uint256 amountShares, uint256 pps, uint256 decimals) internal virtual {
    usersAccumulatorShares[user] += amountShares;
    usersAccumulatorCostBasis[user] += Math.mulDiv(amountShares, pps , 10 ** decimals);
}
```

Then when computing the fee in `OUTFLOW` hook:

```
uint256 costBasis = calculateCostBasisView(user, usedShares);
feeAmount = _calculateFees(costBasis, amountAssets, feePercent);
```

The costBasis is inflated. If costBasis is inflated, then the fee amount is 0 when `costBasis > amountAssets`:

```
uint256 profit = amountAssets > costBasis ? amountAssets - costBasis : 0;
if (profit > 0) {
    if (feePercent == 0) revert FEE_NOT_SET();
    feeAmount = Math.mulDiv(profit, feePercent, 10_000);
}
```

By using untrusted yield source, the user can follow the execution flow to bypass the fee and tamper `SuperLedger` shares and accumulator cost accounting.

Recommendation:

1. Setting yield source address in SuperLedgerConfig `YieldSourceOracleConfig`.
2. Tracking the share and accumulator cost by yield source and user:

```
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);
    }
```

Superform: Fixed #2 in PR 317. We acknowledge that fees can be bypassed by specifying untrusted yield sources or ledgers, which is intended.

Cantina Managed: Verified, the ledger now identifies `shares` and `costBasis` by the user and yield source.

3.2.2 Decoding in `Swap1InchHook._validateGenericSwap()` is not compatible with 1inch's `AggregationRouterV6`

Severity: Medium Risk

Context: Swap1InchHook.sol#L147-L150, AggregationRouterV6.mainnet.sol.

Description: In `Swap1InchHook._validateGenericSwap()`, calldata for `I1InchAggregationRouterV6.swap()` is decoded as follows:

```
//swap(IAggregationExecutor executor, SwapDescription calldata desc, bytes calldata permit, bytes calldata
↪  data)
// external payable
(, I1InchAggregationRouterV6.SwapDescription memory swapDescription,,) =
    abi.decode(txData_, (IAggregationExecutor, I1InchAggregationRouterV6.SwapDescription, bytes, bytes));
```

However, `AggregationRouterV6.swap()` does not have the `bytes permit` parameter:

```
function swap(
    IAggregationExecutor executor,
    SwapDescription calldata desc,
    bytes calldata data
)
```

As a result, the decoding in `_validateGenericSwap()` will fail, making it impossible to perform generic swaps with `Swap1InchHook`.

Recommendation: Correct the decoding of `I1InchAggregationRouterV6.swap()` as such:

```
- (, I1InchAggregationRouterV6.SwapDescription memory swapDescription,,) =
-     abi.decode(txData_, (IAggregationExecutor, I1InchAggregationRouterV6.SwapDescription, bytes, bytes));
+ (, I1InchAggregationRouterV6.SwapDescription memory swapDescription,) =
+     abi.decode(txData_, (IAggregationExecutor, I1InchAggregationRouterV6.SwapDescription, bytes));
```

Superform: Fixed in PR 320.

Cantina Managed: Verified, the recommended fix was implemented.

3.2.3 `Swap1InchHook._getBalance()` is not compatible with native assets swaps through Clipper exchange

Severity: Medium Risk

Context: Swap1InchHook.sol#L28, Swap1InchHook.sol#L232-L240, AggregationRouterV6.mainnet.sol#L4565, AggregationRouterV6.mainnet.sol#L4653

Description: In `Swap1InchHook`, native assets are represented with the `0xEee...` address:

```
address constant NATIVE = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
```

In `_getBalance()`, if the output token (i.e. `dstToken`) is the native asset address, it returns the native asset balance of `dstReceiver` instead of their ERC20 balance:

```
function _getBalance(bytes calldata data) private view returns (uint256) {
    address dstToken = address(bytes20(data[:20]));
    address dstReceiver = address(bytes20(data[20:40]));

    if (dstToken == NATIVE) {
        return dstReceiver.balance;
    }
    return IERC20(dstToken).balanceOf(dstReceiver);
}
```

However, for Clipper exchange, native assets are represented with `address(0)` instead of `0xEee...`:

```
IERC20 private constant _ETH = IERC20(address(0));
```

```
} else if (dstToken == _ETH) {
```

As a result, if `Swap1InchHook` is used to swap through Clipper exchange with the destination token as native assets, the call to `preExecute()` will revert as `_getBalance()` ends up calling `balanceOf()` on the zero address.

Recommendation: In `_getBalance()`, check for the zero address alongside `0xEee...`:

```
- if (dstToken == NATIVE) {
+ if (dstToken == NATIVE || dstToken == address(0)) {
      return dstReceiver.balance;
  }
```

Superform: Fixed in PR 321.

Cantina Managed: Verified, the recommended fix was implemented.

3.2.4 Calling `underlying_coins()` in `Swap1InchHook._validateUnoswap()` restricts which Curve pools can be used

Severity: Medium Risk

Context: Swap1InchHook.sol#L190-L194, AggregationRouterV6.mainnet.sol#L5479, AggregationRouterV6.mainnet.sol#5449, AggregationRouterV6.mainnet.sol#L5382

Description: When `Swap1InchHook` is used to swap through Curve, `_validateUnoswap()` fetches the output token (i.e. `dstToken` below) as such:

```
if (protocol == ProtocolLib.Protocol.Curve) {
    // CURVE
    uint256 dstTokenIndex = (Address.unwrap(dex) >> _CURVE_TO_COINS_ARG_OFFSET) & _CURVE_TO_COINS_ARG_MASK;
    dstToken = ICurvePool(dex.get()).underlying_coins(int128(uint128(dstTokenIndex)));
} else {
```

However, calling `underlying_coins()` only works for Curve pools which have this function, which are lending pools according to Curve's documentation. 1inch determines the output token from a swap as follows:

```
let toToken
{  // Stack too deep
    let toSelectorOffset := and(shr(_CURVE_TO_COINS_SELECTOR_OFFSET, dex), _CURVE_TO_COINS_SELECTOR_MASK)
    let toTokenIndex := and(shr(_CURVE_TO_COINS_ARG_OFFSET, dex), _CURVE_TO_COINS_ARG_MASK)
    toToken := curveCoins(pool, toSelectorOffset, toTokenIndex)
}
```

```
function curveCoins(pool, selectorOffset, index) -> coin {
    mstore(0, _CURVE_COINS_SELECTORS)
    mstore(add(selectorOffset, 4), index)
    if iszero(staticcall(gas(), pool, selectorOffset, 0x24, 0, 0x20)) {
        reRevert()
    }
    coin := mload(0)
}
```

```
// Curve Pool function selectors for different `coins` methods. For details, see
↪  contracts/interfaces/ICurvePool.sol
bytes32 private constant _CURVE_COINS_SELECTORS =
↪  0x87cb4f5723746eb8c6610657b739953eb9947eb0000000000000000000000000;
```

Essentially, the code gets the selector offset through:

```
(dex >> _CURVE_TO_COINS_SELECTOR_OFFSET) & _CURVE_TO_COINS_SELECTOR_MASK
```

Afterwards, the selector offset is used to extract the appropriate Curve function from `_CURVE_COINS_SELECTORS`. The functions in `_CURVE_COINS_SELECTORS` are:

```
base_coins(uint256)
coins(int128)
coins(uint256)
underlying_coins(int128)
underlying_coins(uint256)
```

As a result, Curve pools which have the functions above (excluding `underlying_coins(int128)`) are incompatible with `Swap1InchHook`, even though 1inch itself supports them.

Recommendation: Consider refactoring `_validateUnoswap()` to fetch `dstToken` as such:

```
uint256 private constant _CURVE_TO_COINS_SELECTOR_OFFSET = 208;
uint256 private constant _CURVE_TO_COINS_SELECTOR_MASK = 0xff;
uint256 private constant _CURVE_TO_COINS_ARG_OFFSET = 216;
uint256 private constant _CURVE_TO_COINS_ARG_MASK = 0xff;
```

```
uint256 selectorOffset = (Address.unwrap(dex) >> _CURVE_TO_COINS_SELECTOR_OFFSET) &
↪  _CURVE_TO_COINS_SELECTOR_MASK;
uint256 dstTokenIndex = (Address.unwrap(dex) >> _CURVE_TO_COINS_ARG_OFFSET) & _CURVE_TO_COINS_ARG_MASK;

address dstToken;
if (selectorOffset == 0) {
    dstToken = ICurvePool(dex.get()).base_coins(dstTokenIndex);
} else if (selectorOffset == 4) {
    dstToken = ICurvePool(dex.get()).coins(int128(uint128(dstTokenIndex)));
} else if (selectorOffset == 8) {
    dstToken = ICurvePool(dex.get()).coins(dstTokenIndex);
}  else if (selectorOffset == 12) {
    dstToken = ICurvePool(dex.get()).underlying_coins(int128(uint128(dstTokenIndex)));
} else if (selectorOffset == 16) {
    dstToken = ICurvePool(dex.get()).underlying_coins(dstTokenIndex);
} else {
    revert INVALID_SELECTOR_OFFSET();
}
```

The gist 52d472190 showcases a test that can be used as a reference to ensure the logic above matches the code in `AggregationRouterV6`.

Superform: Fixed in PR 324.

Cantina Managed: Verified, the recommended fix was implemented.

3.2.5 Missing sequencer uptime check for L2s in `SuperOracle`

Severity: Medium Risk

Context: SuperOracle.sol#L230-L230

Description: In `SuperOracle`, `_getQuoteFromOracle()` fetches token prices using Chainlink's price feeds:

```
(, int256 answer,, uint256 updatedAt,) = AggregatorV3Interface(oracle).latestRoundData();
```

However, the function does not check if the sequencer is up before using the price returned by Chainlink. This is required on L2s as price feeds could be inaccurate when the sequencer is down, which is described in Chainlink's documentation.

Recommendation: Consider having a separate `SuperOracle` contract meant to be deployed on L2s. For example, a `SuperOracleL2` contract could inherit `SuperOracle`. In `SuperOracleL2`, `_getQuoteFromOracle()` should be overridden to include a L2 sequencer check, as shown in this example.

Superform: Fixed in PR 351.

Cantina Managed: Verified, the recommendation has been implemented.

3.2.6 Staleness periods should be configurable for each Chainlink price feed in `SuperOracle`

Severity: Medium Risk

Context: SuperOracle.sol#L23-L24, SuperOracle.sol#L230-L236

Description: In `SuperOracle`, the staleness period is stored for each provider:

```
/// @notice Mapping of provider to max staleness period
mapping(uint256 provider => uint256 maxStaleness) public providerMaxStaleness;
```

In `_getQuoteFromOracle()`, this staleness period is then used to check the price returned from Chainlink's price feed is not stale:

```
(, int256 answer,, uint256 updatedAt,) = AggregatorV3Interface(oracle).latestRoundData();

// Validate data
if (answer <= 0 || block.timestamp - updatedAt > providerMaxStaleness[provider]) {
    if (revertOnError) revert ORACLE_UNTRUSTED_DATA();
    return 0;
}
```

However, the staleness period should be configured for each **oracle**, not each provider, since different Chainlink feeds have different staleness periods. For example:

- ETH / USD has a staleness period of 1 hour (3600 seconds).
- SOL / USD has a staleness period of 1 day (86400 seconds).

Recommendation: Consider refactoring `SuperOracle` to store staleness periods based on oracle addresses, rather than based on providers.

Superform: Fixed in PR 351.

Cantina Managed: Verified, the staleness period can now be configured for each individual price oracle.

3.2.7 Exact balance check in `SuperExecutor._performErc20FeeTransfer()` prevents collecting rebasing tokens as fees

Severity: Medium Risk

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

Context: SuperExecutor.sol#L152-L161.

Description: `SuperExecutor._performErc20FeeTransfer()` checks that the token balance of the `feeRecipient` increased by exactly `feeAmount`:

```
uint256 balanceBefore = IERC20(assetToken).balanceOf(feeRecipient);
Execution[] memory feeExecution = // ...
_execute(account, feeExecution);
uint256 balanceAfter = IERC20(assetToken).balanceOf(feeRecipient);
if (balanceAfter - balanceBefore != feeAmount) revert FEE_NOT_TRANSFERRED();
```

However, this check will revert for rebasing tokens due to their internal share mechanics, the most prominent being stETH. stETH transfers will sometimes result in 1-2 wei less stETH received, as described in Lido's documentation.

Proof of Concept:

```
contract StEthTest is Test {
    IStEth ST_ETH = IStEth(0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84);
    address WST_ETH = 0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84;

    function test_transferRounding() public {
        address to = makeAddr("to");

        vm.prank(WST_ETH);
        ST_ETH.transfer(to, 1e18);

        assertEq(ST_ETH.balanceOf(to), 1e18 - 1);
    }
}

interface IStEth {
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 value) external returns (bool);
}
```

As a result, it is not possible to collect fees for rebasing tokens in `SuperExecutor`. Another prominent rebasing token would be AAVE's aTokens, which is commonly used in staking and yield vaults.

Recommendation: In `SuperExecutor._performErc20FeeTransfer()`, consider removing the balance check or modifying it to allow deviations of a few wei.

Superform: Fixed in PR 331.

Cantina Managed: Verified, the check has been modified to allow a deviation of up to 10% of the expected fee amount.

3.2.8 Incorrect deposit check in `SuperNativePaymaster._validatePaymasterUserOp()` will always revert

Severity: Medium Risk

Context: SuperNativePaymaster.sol#L91-L93, EntryPoint.sol#L624-L629

Description: In `SuperNativePaymaster`, `_validatePaymasterUserOp()` checks that its current deposit in the `Entrypoint` is not less than the maximum gas cost of the transaction to be sponsored (i.e. `maxCost` below):

```
if (entryPoint.getDepositInfo(address(this)).deposit < maxCost) {
    revert INSUFFICIENT_BALANCE();
}
```

However, in the `Entrypoint` contract, `requiredPreFund` is subtracted from the paymaster's deposit *before* calling `validatePaymasterUserOp()`:

```
uint256 requiredPreFund = opInfo.prefund;
if (!_tryDecrementDeposit(paymaster, requiredPreFund)) {
    revert FailedOp(opIndex, "AA31 paymaster deposit too low");
}
uint256 pmVerificationGasLimit = mUserOp.paymasterVerificationGasLimit;
(context, validationData) = _callValidatePaymasterUserOp(opIndex, op, opInfo);
```

Note that `requiredPreFund` is eventually passed to `validatePaymasterUserOp()` as `maxCost`. Since `SuperNativePaymaster` should not hold any additional deposits apart from the amount needed for the current execution, the check mentioned above will always revert when trying to perform the last execution as `getDepositInfo(address(this))` will be `0`.

For example:

- Assume `SuperNativePaymaster` has no balance and deposits.
- It is used to execute a call which has `requiredPreFund = maxCost = 0.1 ETH`.
- When `handleOps()` is called, 0.1 ETH is transferred to the `Entrypoint` and added to the contract's deposits.
- In the `Entrypoint`:
  - `_tryDecrementDeposit()` is called, which decreases the contract's deposits from 0.1 ETH to 0.
  - `validatePaymasterUserOp()` is called.
  - The `getDepositInfo(address(this)) < maxCost` check reverts as `maxCost = 0.1 ETH`, but the contract's deposit is `0`.

As a result, `SuperNativePaymaster` cannot be used to sponsor user transactions.

Recommendation: Consider removing the `entryPoint.getDepositInfo(address(this)).deposit < maxCost` check from `_validatePaymasterUserOp()`.

Superform: Fixed in PR 333.

Cantina Managed: Verified, the issue has been fixed by removing the check.

3.2.9 Malicious relayer can spoof `AcrossTargetExecutor.handleV3AcrossMessage()` to grief cross-chain transactions

Severity: Medium Risk

Context: AcrossTargetExecutor.sol#L129-L132

Description: According to Across V3's documentation, the protocol does not guarantee the integrity of the parameters that `handleV3AcrossMessage()` is called with:

> Avoid making unvalidated assumptions about the `message` data supplied to `handleV3AcrossMessage()`. Across does not guarantee message integrity, only that a relayer who spoofs a message will not be repaid by Across. If integrity is required, integrators should consider including a depositor signature in the message for additional verification. Message data should otherwise be treated as spoofable and untrusted for use beyond directing the funds passed along with it.

More specifically, a malicious relayer can call `handleV3AcrossMessage()` with any arguments, regardless of what was sent by users in the cross-chain transaction on the source chain. As such, all parameters of `handleV3AcrossMessage()` must be validated to ensure they are not manipulated by a malicious relayer.

However, in `AcrossTargetExecutor.handleV3AcrossMessage()`, only the following parameters are verified by the user's signature:

```
// @dev validate execution
bytes memory destinationData = abi.encode(_nonce, executorCalldata, uint64(block.chainid), account);
bytes4 validationResult = IValidator(superDestinationValidator).isValidSignatureWithSender(account,
↪  bytes32(0), abi.encode(sigData, destinationData));
if (validationResult != SIGNATURE_MAGIC_VALUE) revert INVALID_SIGNATURE();
```

The inputs/parameters which are not verified are:

- `tokenSent` and `amount`.
- `initData`, `account` and `intentAmount` decoded from `message`.

Not including `initData` and `account` in the user's signature is safe as changing `initData`/`account` would result in a different `account` address and would not affect the user's cross-chain transaction. However, a malicious relayer can manipulate `tokenSent`, `amount` and `intentAmount` to influence the tokens transferred to the account and whether `executorCalldata` is executed. For example:

- User sends a transaction on the source chain to transfer 1000 USDC and deposit it into a vault. This means:
- `tokenSent` should be USDC.
- `amount` and `intentAmount` should be `1000e6`.
- `executorCalldata` contains calldata to deposit the USDC into a vault.
- On the destination chain, a malicious relayer calls `handleV3AcrossMessage()` with the following parameters:
- `tokenSent` as USDC.
- `amount = 0`.
- `intentAmount = uint256.max`.
- When `handleV3AcrossMessage()` is executed:
  - No tokens are transferred to the account as `amount = 0`.
  - Due to this `intentAmount` check, `executionCalldata` is skipped and not executed.
  - The nonce is consumed, therefore the user's original transaction can no longer be relayed.
- Since the cross-chain transaction cannot be relayed, a refund occurs on the source chain to the user's account.

As seen from above, by manipulating `tokenSent`, `amount` and `intentAmount`, a malicious relayer can block cross-chain transactions from being relayed.

Recommendation: Consider including `tokenSent` and `intentAmount` in `destinationData`:

```
  // @dev validate execution
- bytes memory destinationData = abi.encode(_nonce, executorCalldata, uint64(block.chainid), account);
+ bytes memory destinationData = abi.encode(_nonce, executorCalldata, uint64(block.chainid), account,
↪  tokenSent, intentAmount);
```

Additionally, when the amount of tokens transferred to the account is less than `intentAmount`, `handleV3AcrossMessage()` should revert instead of returning:

```
  // @dev check if the account has sufficient balance before proceeding
  if (intentAmount != 0 && token.balanceOf(account) < intentAmount) {
-     emit AcrossTargetExecutorReceivedButNotEnoughBalance(account);
-     return;
+     revert AcrossTargetExecutorReceivedButNotEnoughBalance(account);
  }
```

This prevents `handleV3AcrossMessage()` from being called with an `amount` less than `intentAmount`.

Superform: Fixed in PR 350.

Cantina Managed: Verified, `tokenSent` and `intentAmount` have been added to `destinationData`.

Additionally, the nonce is only increased when `executorCalldata` is executed. While this fixes the issue of `handleV3AcrossMessage()` being called with less tokens sent than expected, this allows a message to be relayed multiple times for a same merkle leaf. As such, a signature can be replayed as long as the attached calldata has not been executed.

3.3 Low Risk

3.3.1 `ApproveAndSwapOdosHook.sol` should handle native balance change properly

Severity: Low Risk

Context: ApproveAndSwapOdosHook.sol#L101-L104

Description: `ApproveAndSwapOdosHook.sol` should handle native balance change properly. The `SwapOdosHook.sol` does track the native token balance change properly. Otherwise, if the destination token is native ETH, the output amount will be 0.

Recommendation:

```
function _getBalance(address account, bytes memory data) private view returns (uint256) {
    address outputToken = BytesLib.toAddress(BytesLib.slice(data, 72, 20), 0);

    if (outputToken == address(0)) {
        return account.balance;
    }

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

Superform: Fixed in PR 308.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.2 Consider adding reentrancy protection for `_processHook` in `SuperExecutor.sol`

Severity: Low Risk

Context: SuperExecutor.sol#L90-L101

Description: The `_execute` make an external call and perform asset transfer such as `deposit` or `redeem`, then the accounting is updated and the code charge fee. The code does not follow the `check effect pattern` because the code modify the internal state after the external call.

Recommendation: Consider using OpenZeppelin's reentrancy guard:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
```

and

```
function _processHook(address account, ISuperHook hook, address prevHook, bytes memory hookData) private
↪  nonReentrant {
```

Superform: Fixed in PR 309.

Cantina Managed: Verified, the `nonReentrant` modifier was added to `_processHook()`.

3.3.3 Malicious actor can front-run the `yieldSourceOracleId` setting

Severity: Low Risk

Context: SuperLedgerConfiguration.sol#L111-L119

Description: In `SuperLedgerConfiguration`, `setYieldSourceOracles()` is permissionless and can be called by anyone. As such, if a yield source oracle is not initialized, a malicious attacker could front-run a legitimate call to `setYieldSourceOracles()` to set the configuration at the same `oracleId` first. For example:

1. Alice wants to set the `yieldSourceOracle` and `ledger`.
2. Bob front-runs Alice's `setYieldSourceOracles` transaction and becomes the manager.
3. Bob can set `yieldSourceOracle` and `ledger` to a malicious address.

A malicious ledger contract could return a very high fee (up to the 50% max fee limit) to steal funds from users:

```
// Update accounting and get fee amount if any
uint256 feeAmount = ISuperLedger(config.ledger).updateAccounting(
    account,
    yieldSource,
    yieldSourceOracleId,
    _type == ISuperHook.HookType.INFLOW,
    ISuperHookResult(address(hook)).outAmount(),
    ISuperHookResultOutflow(address(hook)).usedShares()
);
```

Even if a yield source oracle is initialized, the manager can front-run a user's `OUTFLOW` hook execution by changing the `yieldSourceOracle` and `ledger` contract to force the user pay high fees. For example:

- `feePercent` is 1%.
- A user signs and executes an outflow, thinking that the current profit fee of 1% is acceptable.
- Yield source manager increases `feePercent` to 50%.
- When `SuperExecutor.execute()` is called, the user pays a 50% fee instead of 1%.

This effectively means the user has no control over the fee he pays when an outflow occurs; it entirely depends on what `YieldSourceOracleConfig.feePercent` is set to whenever he executes an outflow.

Recommendation: Consider the following mitigations:

- Add a time lock queue to a manager changing parameters of a yield source oracle, ledger address and fee percent.
- Allow users to specify the expected fee or a fee slippage in `hookData`, which is checked in `_updateAccounting()`. If the current fee exceeds the maximum fee the user is willing to pay, revert the transaction.

Additionally, while it is the responsibility of accounts/users to ensure they never pass a malicious `yieldOracleId` to `SuperExecutor`, it is probably best to check that the `oracleId` being recommended to users off-chain is actually owned by the protocol.

Superform: Fixed in PR 315.

Cantina Managed: Verified, a timelock of one week has been introduced when managers are changing the parameters of a yield source.

3.3.4 Approval pattern in hooks is not compatible with tokens which revert on zero allowance

Severity: Low Risk

Context: ApproveERC20Hook.sol#L50-L53, ApproveAndDeposit4626VaultHook.sol#L57-L65, ApproveAndRequestDeposit7540VaultHook.sol#L66-L77

Description: In all hooks, ERC20 token allowances are set by performing a zero-approval before calling `approve()` with the allowance amount. This pattern can be seen in `ApproveERC20Hook`:

```
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)) });
```

Similarly, for hooks calling protocol functions which require a prior token allowance, a call to `approve(..., 0)` is performed after the action to reset the allowance from the account to the protocol. For example, in `ApproveAndDeposit4626VaultHook`:

```
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(IERC4626.deposit, (amount, account)) });
executions[3] =
    Execution({ target: token, value: 0, callData: abi.encodeCall(IERC20.approve, (yieldSource, 0)) });
```

However, calling `approve(..., 0)` does not work for certain tokens that revert when `approve()` is called with zero allowance. BNB on mainnet is one such token:

```
/* Allow another contract to spend some tokens in your behalf */
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

As a result, any hook which performs an approval cannot be used for BNB as the call to `approve()` will always revert.

Recommendation: In all hooks with approvals, document that they are not compatible with tokens which revert on zero approvals, especially BNB on mainnet. Additionally, consider introducing separate hooks specifically for handling such tokens if there is a need to.

Superform: Acknowledged, added comments in PR 323 to document this issue.

Cantina Managed: Acknowledged.

3.3.5 Incorrect `maxGasLimit` check and calculation in `SuperNativePaymaster._validatePaymasterUserOp()`

Severity: Low Risk

Context: SuperNativePaymaster.sol#L106-L109

Description: In `SuperNativePaymaster._validatePaymasterUserOp()`, `maxGasLimit` is calculated and checked to be below the sum of `verificationGasLimit` and `callGasLimit` as such:

```
// verification + call  gas limit <= maxGasLimit
uint256 totalGasLimit =
    PaymasterGasCalculator.getVerificationGasLimit(userOp) + PaymasterGasCalculator.getCallGasLimit(userOp);
if (maxGasLimit > totalGasLimit || maxGasLimit == 0) revert INVALID_MAX_GAS_LIMIT();
```

However, there are two issues with this check:

1. The check should be `maxGasLimit < totalGasLimit` as it doesn't make sense to enforce that `maxGasLimit` is *less* than the maximum amounts of gas used. This would also match the comment.
2. Paymaster and bundler gas limits are wrongly excluded from `totalGasLimit`. With reference to the `Entrypoint` contract, the other gas limits are:
    - `paymasterVerificationGasLimit` - the gas limit for `validatePaymasterUserOp()`.
    - `paymasterPostOpGasLimit` - the gas limit for `postOp()`.
    - `preVerificationGas` - gas paid to the bundler.

The sum of these three limits alongside `verificationGasLimit` and `callGasLimit` is the total amount of gas for execution.

Assuming `maxGasLimit` is the amount of gas the user pays fees for, the fees charged to the user should also include gas for the paymaster and bundler. However, the current calculation excludes paymaster and bundler gas from fees.

Recommendation: Correct the check as such:

```
- if (maxGasLimit > totalGasLimit || maxGasLimit == 0) revert INVALID_MAX_GAS_LIMIT();
+ if (maxGasLimit < totalGasLimit) revert INVALID_MAX_GAS_LIMIT();
```

Additionally, include `paymasterVerificationGasLimit`, `paymasterPostOpGasLimit` and `preVerificationGas` in the calculation of `totalGasLimit`.

Superform: Fixed in PR 333.

Cantina Managed: Verified, the `maxGasLimit` check has been removed.

3.3.6 `SuperMerkleValidator._isSignatureValid()` does not revert for errors unrelated to signature verification

Severity: Low Risk

Context: SuperMerkleValidator.sol#L235-L239

Description: The ERC 4337 specification states:

> MUST validate that the signature is a valid signature of the `userOpHash`, and SHOULD return `SIG_VALIDATION_FAILED` (1) without reverting on signature mismatch. Any other error MUST revert.

More specifically, "Any other error MUST revert". However, this isn't adhered to in the current implementation of `SuperMerkleValidator._isSignatureValid()`, as shown below:

```
if (proof.length == 0) return false;

// Verify merkle proof
bool isValid = MerkleProof.verify(proof, merkleRoot, leaf);
return isValid && signer == accountOwners[sender] && validUntil >= block.timestamp;
```

When the provided merkle proof is not valid (i.e. `MerkleProof.verify()`), the function should revert instead of returning `false`.

Additionally:

1. Providing merkle proofs with a length of `0` (i.e. `proof.length == 0`) should be allowed in order to prove merkle trees with one leaf.
2. The `validUntil >= block.timestamp` check is not needed as it is already checked in the `Entrypoint` contract.

Recommendation: Modify the logic in `_isSignatureValid()` to:

```
// Verify merkle proof
if (!MerkleProof.verify(proof, merkleRoot, leaf)) revert INVALID_PROOF();

return signer == accountOwners[sender];
```

The `proof.length != 0` and `validUntil >= block.timestamp` checks have been excluded, as mentioned above.

Superform: Fixed in commit 0d998cd.

Cantina Managed: Verified, `_processSignatureAndVerifyLeaf()` reverts if the merkle proof is invalid.

3.3.7 `spToken` is wrongly set to the vault address for ERC 7540 hooks

Severity: Low Risk

Context: Deposit7540VaultHook.sol#L70, Withdraw7540VaultHook.solL#78

Description: According to ERC 7540, the vault address may not be the share token:

> Smart contracts implementing this Vault standard MUST implement the ERC 7575 standard (in particular the `share` method).

However, in the `preExecute()` hook of `Deposit7540VaultHook` and `Withdraw7540VaultHook` respectively, `spToken` is set to the yield source address:

```
spToken = data.extractYieldSource();
```

```
spToken = yieldSource;
```

Since `spToken` should be a token that can be locked in the future in a treasury, it should be the address at `share()` instead.

Recommendation: Modify `Deposit7540VaultHook.preExecute()` to fetch the share token address as such:

```
- spToken = data.extractYieldSource();
+ spToken = IERC7540(data.extractYieldSource()).share();
```

Similarly, for `Withdraw7540VaultHook`:

```
- spToken = yieldSource;
+ spToken = IERC7540(yieldSource).share();
```

Superform: Fixed in PR 325.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.8 ERC20 hooks could silently fail for no-revert-on-failure tokens

Severity: Low Risk

Context: ApproveERC20Hook.sol#L50-L53, TransferERC20Hook.sol#L49-L50

Description: `ApproveERC20Hook` calls `IERC20.approve()` through a low-level call via an account's batch execution:

```
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)) });
```

Similarly, `TransferERC20Hook` calls `IERC20.transfer()` through batch execution:

```
executions = new Execution[](1);
executions[0] = Execution({ target: token, value: 0, callData: abi.encodeCall(IERC20.transfer, (to, amount)) });
```

However, since `SuperExecutor` does not check the return values of `approve()`/`transfer()` after execution, it is possible for executions involving `ApproveERC20Hook`/`TransferERC20Hook` to incorrectly succeed for no-revert-on-failure tokens. More specifically, the call to `approve()`/`transfer()` fails and returns `false` instead of reverting, which will not be checked by `SuperExecutor`.

This could be dangerous if `ApproveERC20Hook`/`TransferERC20Hook` is composed with other hooks which rely on its `outAmount`, as the approval/transfer could silently fail and `outAmount` would be set to `0` instead of the intended amount by the owner.

Recommendation: Consider documenting that no-revert-on-failure tokens are not supported by `ApproveERC20Hook` and `TransferERC20Hook` if it is an acceptable limitation. Otherwise, `SuperExecutor` would have to be refactored to pass return values to hooks in order for `ApproveERC20Hook`/`TransferERC20Hook` to check them.

Superform: Acknowledged, added comments in PR 328 to document that we do not support such tokens.

Cantina Managed: Acknowledged.

3.3.9 `pendingUpdate.timestamp != 0` check in `SuperOracle.queueOracleUpdate()` is dangerous

Severity: Low Risk

Context: SuperOracle.sol#L144

Description: In `SuperOracle`, `queueOracleUpdate()` checks that `pendingUpdate.timestamp` is `0`, which prevents the owner from overwriting a pending update if one exists:

```
if (pendingUpdate.timestamp != 0) revert PENDING_UPDATE_EXISTS();
```

However, this check is dangerous for two reasons:

1. If `queueOracleUpdate()` is ever called to queue an incorrect update, there is no way for the owner to "cancel" that update. The wrong update must and will probably be executed through `executeOracleUpdate()`.
2. If `executeOracleUpdate()` always reverts for some reason, there is a risk of `queueOracleUpdate()` becoming un-callable and the owner will never be able to update `usdQuotedOracle`. However, note that this is currently not possible since `executeOracleUpdate()` has the exact same checks as `queueOracleUpdate()`.

Recommendation: Consider removing the `pendingUpdate.timestamp != 0` check from `queueOracleUpdate()` to allow owners to overwrite the current pending update.

Superform: Fixed in PR 329.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.10 Consider try catch the `execution` in `handleV3AcrossMessage` after token transfer

Severity: Low Risk

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

Description: In `handleV3AcrossMessage`, after token is sent to the smart account, the code can execute a sequence of action via hook:

```
// @dev _execute -> executeFromExecutor -> SuperExecutorBase.execute
Execution[] memory execs = new Execution[](1);
execs[0] = Execution({
    target: address(this),
    value: 0,
    callData: executorCalldata
});
```

However, if the sequence of action fails (for example, making a swap but revert because of slippage), a refund is triggered (see the docs):

> If the recipient contract's `handleV3AcrossMessage` function reverts when `message`, `tokenSent`, `amount`, and `relayer` are passed to it, then the fill on destination will fail and cannot occur. In this case the deposit will expire when the destination `SpokePool` timestamp exceeds the deposit `fillDeadline` timestamp, the depositor will be refunded on the `originChainId`. Ensure that the depositor address on the origin `SpokePool` is capable of receiving refunds.

Recommendation: The recommendation is just try catch the execution to make sure a bridge refund is not triggered because of revert in hook action.

Superform: Fixed in PR 334.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.11 `AcrossTargetExecutor` address should be included in `destinationData`

Severity: Low Risk

Context: AcrossTargetExecutor.sol#L129-L132

Description: In `handleV3AcrossMessage()`, the address of `AcrossTargetExecutor` is not included in `destinationData`, which is used to create the merkle leaf signed by the account's owner:

```
// @dev validate execution
bytes memory destinationData = abi.encode(_nonce, executorCalldata, uint64(block.chainid), account);
bytes4 validationResult = IValidator(superDestinationValidator).isValidSignatureWithSender(account,
↪  bytes32(0), abi.encode(sigData, destinationData));
if (validationResult != SIGNATURE_MAGIC_VALUE) revert INVALID_SIGNATURE();
```

However, not including the address of `AcrossTargetExecutor` allows the owner's signature to be replayed if the account mistakenly installs two `AcrossTargetExecutor` contracts as executor modules. This is because there would be two different `nonces` mapping on both contracts.

Recommnedation: Include the address of `AcrossTargetExecutor` (i.e. `address(this)`) in `destinationData`.

Superform: Fixed in PR 340 and PR 366.

Cantina Managed: Verified, `address(this)` has been added to `destinationData`.

3.3.12 `outAmount` of previous hook cannot be used for native assets in `AcrossSendFundsAndExecuteOnDstHook`

Severity: Low Risk

Context: AcrossSendFundsAndExecuteOnDstHook.sol#L84-L86, SpokePool.sol#L1360

Description: In `AcrossSendFundsAndExecuteOnDstHook.build()`, only `inputAmount` is set to `outAmount` when `usePrevHookAmount` is `true`:

```
if (acrossV3DepositAndExecuteData.usePrevHookAmount) {
    acrossV3DepositAndExecuteData.inputAmount = ISuperHookResult(prevHook).outAmount();
}
```

However, this prevents using `outAmount` from the previous hook if the asset to transfer (i.e. `inputToken`) is native asset, since `acrossV3DepositAndExecuteData.value` is not set to `outAmount` as well.

Recommendation: When `usePrevHookAmount` is `true`, consider setting `acrossV3DepositAndExecuteData.value` to `outAmount` if native asset is transferred:

```
  if (acrossV3DepositAndExecuteData.usePrevHookAmount) {
      acrossV3DepositAndExecuteData.inputAmount = ISuperHookResult(prevHook).outAmount();
+     if (
+         acrossV3DepositAndExecuteData.inputToken == address(spokePoolV3.wrappedNativeToken())
+         && acrossV3DepositAndExecuteData.value != 0
+     ) {
+         acrossV3DepositAndExecuteData.value = ISuperHookResult(prevHook).outAmount();
+     }
  }
```

Across V3 determines if the asset to transfer is native asset if `inputToken` is the wrapped native token (e.g. WETH) and `msg.value` is non-zero, as seen in SpokePool.sol#L1360.

Superform: Fixed in PR 341.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.13 Minor errors in hooks

Severity: Low Risk

Context: *(See each case below)*

Description/Recommendation:

1. FluidUnstakeHook.sol#L75 - In `preExecute()`, `lockForSP` is decoded from offset 57. However, it should be decoded from offset 77 instead as 55 is in the middle of `amount`:

   ```
   - lockForSP = _decodeBool(data, 57);
   + lockForSP = _decodeBool(data, 77);
   ```

2. ApproveAndGearboxStakeHook.sol#L52 - In `build()`, the `amount == 0` check should be after `usePrevHookAmount`:

```
- if (amount == 0) revert AMOUNT_NOT_VALID();

if (usePrevHookAmount) {
    amount = ISuperHookResult(prevHook).outAmount();
}
+ if (amount == 0) revert AMOUNT_NOT_VALID();
```

3. ApproveAndSwapOdosHook.sol#L71-L77 - It's not possible for native assets to be swapped through `ApproveAndSwapOdosHook` as the hook ends up calling `approve()` on the zero address. As such, `OdosRouterV2.swap()` should always be called with zero value:

```
executions[2] = Execution({
    target: address(odosRouterV2),
-     value: inputToken == address(0) ? inputAmount : 0,
+     value: 0,
    callData: abi.encodeCall(
        IOdosRouterV2.swap, (_getSwapInfo(account, prevHook, data), pathDefinition, executor,
        ↪  referralCode)
    )
});
```

Superform: Fixed in PR 342.

Cantina Managed: Verified, the recommended fixes were implemented.

3.3.14 Incorrect price per share in `FluidYieldSourceOracle.getPricePerShare()`

Severity: Low Risk

Context: FluidYieldSourceOracle.sol#L23-L26, FluidYieldSourceOracle.sol#L51-L63, FluidYieldSourceOracle.sol#L65-L70

Description: In `FluidYieldSourceOracle.getPricePerShare()`, `rewardPerToken()` is assumed to be the amount of rewards owed to a user per staking token:

```
function getPricePerShare(address yieldSourceAddress) public view override returns (uint256) {
    return IFluidLendingStakingRewards(yieldSourceAddress).rewardPerToken();
}
```

However, in Fluid, `rewardPerToken()` is *not* the amount of rewards per staking token. More specifically, the rewards owed to a user cannot be calculated by `rewardPerToken * stakingTokenAmount`, which `FluidYieldSourceOracle` does. This can be seen in `earned()`, which calculates the rewards owed to a user:

```
/// @notice gets earned reward amount for an `account`, also considering automatic transition to queued next
↪  rewards
function earned(address account) public view returns (uint256) {
    return (_balances[account] * (rewardPerToken() - userRewardPerTokenPaid[account])) / 1e18 +
    ↪  rewards[account];
}
```

Recommendation: Consider the following changes:

- `getTVLByOwnerOfShares()` should call `earned()` to determine the rewards for `ownerOfShares`.
- `getTVL()` should call `getRewardForDuration()` to return the total amount of rewards in the current reward period. Note that this does not exclude rewards which are already claimed.

Alternatively, if the oracle is meant to return the **staking** token balance instead of the reward token balance:

- `decimals()` should return 18.
- `getPricePerShare()` should return `1e18`.
- `getTVLByOwnerOfShares()` should return `IFluidLendingStakingRewards.balanceOf(ownerOfShares)`.
- `getTVL()` should return `IERC20Metadata(yieldSourceAddress).totalSupply()`.

This is because shares in `FluidLendingStakingRewards` are 1:1 to the underlying staking token.

Superform: Fixed in PR 345.

Cantina Managed: Verified, the issue has been addressed by combining `GearboxYieldSourceOracle` and `FluidYieldSourceOracle` into a single `StakingYieldSourceOracle` which returns the staking token balance.

3.3.15 Incorrect TVL calculation in `ERC5115YieldSourceOracle.getTVL()`

Severity: Low Risk

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

Context: ERC5115YieldSourceOracle.sol#L88-L89.

Description: `ERC5115YieldSourceOracle.getTVL()` calculates an ERC 5115 vault's total assets as such:

```
(,, uint8 precision) = yieldSource.assetInfo();
return (totalShares * yieldSource.exchangeRate()) / (10 ** precision);
```

However, as documented in Pendle's `IStandardizedYield`, the total asset balance calculation should divide by `1e18`.

Recommendation: Similar to `getTVLByOwnerOfShares()`, the calculation should be `shares * exchangeRate / 1e18`:

```
- return (totalShares * yieldSource.exchangeRate()) / (10 ** precision);
+ return (totalShares * yieldSource.exchangeRate()) / 1e18;
```

Superform: Fixed in PR 343.

Cantina Managed: Verified, the recommended fix was implemented.

3.3.16 Incorrect reward calculation in `GearboxYieldSourceOracle._getRewardPerToken()`

Severity: Low Risk

Context: GearboxYieldSourceOracle.sol#L76-L78, GearboxYieldSourceOracle.sol#L38-L49, GearboxYieldSourceOracle.sol#L51-L62

Description: In `GearboxYieldSourceOracle._getRewardPerToken()`, `Info.reward` is assumed to be the amount of rewards owed to a user per staking token:

```
function _getRewardPerToken(address yieldSourceAddress) internal view returns (uint256) {
    return uint256(IGearboxFarmingPool(yieldSourceAddress).farmInfo().reward);
}
```

However, `Info.reward` is not the amount of rewards per token. Instead, it is the total amount of rewards to distribute. This can be seen in `FarmAccounting.startFarming()`, where `info.reward` is directly set to the amount of reward tokens (i.e. `amount`).

Recommendation: Consider the following changes:

- `getTVLByOwnerOfShares()` should call `farmed()` to determine the rewards for `ownerOfShares`.
- `getTVL()` should return `farmInfo().reward` as the total amount of rewards.

Alternatively, if the oracle is meant to return the **staking** token balance instead of the reward token balance:

- `decimals()` should return 18.
- `getPricePerShare()` should return `1e18`.
- `getTVLByOwnerOfShares()` should return `IGearboxFarmingPool.balanceOf(ownerOfShares)`.
- `getTVL()` should return `IERC20Metadata(yieldSourceAddress).totalSupply()`.

This is because shares in `FarmingPool` are 1:1 to the underlying staking token.

Superform: Fixed in PR 345.

Cantina Managed: Verified, the issue has been addressed by combining `GearboxYieldSourceOracle` and `FluidYieldSourceOracle` into a single `StakingYieldSourceOracle` which returns the staking token balance.

3.3.17 Not validating `msg.sender` in `preExecute`/`postExecute` creates reentrancy risk

Severity: Low Risk

Context: ApproveERC20Hook.sol#L60-L71

Description: In all hooks, `preExecute()` and `postExecute()` have no access control and do not check `msg.sender`. As such, they can be called directly to manipulate variables in transient storage, such as `outAmount` and `usedShares`, which are commonly used when executing subsequent hooks or fee calculations. This is problematic when execution contains an external callback to an untrusted party, such as:

- A fee in native assets is collected after executing an outflow hook, which perform an external callback to the `feeRecipient`.
- A swap through `Swap1inchHook` has native asset as the output token and an untrusted party as the swap receiver.
- User redeems asset from `ERC4626.sol` and the asset is a ERC777 token or native token, which triggers a callback function to call `hook.preExecute` to reset the `outputAmount`.

When the untrusted party receives the external callback, they can directly call `preExecute()` or `postExecute()` to influence the account's execution. For example, if a subsequent hook has `usePrevHookAmount = true` and uses `outAmount` as its input amount, the attacker can control the hook's input amount by calling `preExecute()` to directly set `outAmount`.

Recommendation: Consider checking `msg.sender` is `SuperExecutor` in all `preExecute()` and `postExecute()` functions.

Superform: Fixed in PR 348.

Cantina Managed: Verified, the issue has been addressed by only allowing the first caller of `preExecute()`/`postExecute()` to call both functions afterwards. Under normal execution, reentrancy is no longer possible as:

- `SuperExecutor` calls `preExecute()` first, so `preExecute()`/`postExecute()` can no longer be called directly as only `SuperExecutor` is allowed to call both functions.
- The hooks cannot be called by re-entering `SuperExecutor` due to the `nonReentrant` modifier on `_processHook()`.

Note that this form of reentrancy protection prevents a hook from being used by different callers in the same transaction, which is not required functionality.

3.3.18 Conflicting functionality in `SuperOracle`

Severity: Low Risk

Context: SuperOracle.sol?lines=145,151, SuperOracle.sol#L72-L73, AbstractYieldSourceOracle.sol#L300, SuperOracle.sol#L65

Description: In `SuperOracle`, there seems to be some confusion between whether `provider` is some form of identifier or an array index. The comment in SuperOracle.sol#L23 and the loop in SuperOracle.sol#L80-L93 in `getQuoteFromProvider()` suggests it is an array and `usdQuotedOracle` is supposed to be a mapping of `address base => address[] oracles`. However, the remaining functions treat `provider` as an arbitrary 140-bit identifier, for example:

- `setProviderMaxStaleness()` can be called with any uint256 value as `provider`.
- `queueOracleUpdate()` allows the provider to be set to any uint256. More importantly, `provider` can be `0` (which would collide with `ORACLE_PROVIDER_AVERAGE`) or more than `MAX_PROVIDERS`.

As a result of this confusion, there are several issues in `SuperOracle`.

1. `queueOracleUpdate()` and `_configureOracles()` check that `providers.length` does not exceed `MAX_PROVIDERS`:

```
uint256 providersLength = providers.length;
if (providersLength > MAX_PROVIDERS) {
    revert MAX_PROVIDERS_EXCEEDED();
}
if (bases.length != providersLength || providersLength != oracleAddresses.length) {
    revert ARRAY_LENGTH_MISMATCH();
}
```

However, this check is wrong for two reasons:

- `MAX_PROVIDERS` isn't actually the maximum number of providers as `executeOracleUpdate()` doesn't clear previous data. This means the owner can add at most 10 providers each time, but it is possible for an unlimited number of providers to be configured.
- Enforcing `providersLength <= MAX_PROVIDERS` means the owner can't configure more than 10 base assets, since `providersLength == bases.length`.

2. In `getQuoteFromProvider()`, the following check should be moved into the `else` branch (i.e. only check `usdQuotedOracle[base][oracleProvider] != 0` when `oracleProvider` is not `ORACLE_PROVIDER_AVERAGE`):

```
address oracle = usdQuotedOracle[base][oracleProvider];
if (oracle == address(0)) revert NO_ORACLES_CONFIGURED();
```

Otherwise, it would wrongly revert when `ORACLE_PROVIDER_AVERAGE` is specified since there shouldn't be an oracle address set for `ORACLE_PROVIDER_AVERAGE`.

3. In `AbstractYieldSourceOracle`, `_encodeProvider()` directly casts `provider` to `uint160`:

```
return address(uint160((provider << 20) | uint160(USD)));
```

If `provider` is an identifier larger than 140 bits, `uint160(provider << 20)` will overflow.

4. `getQuoteFromProvider()` does not ensure the oracle address at `oracleProvider` is actually denominated in `quote` tokens. It's entirely possible for the function to be called with the wrong `quote` address, causing the returned price to have wrong decimals. Additionally, the README mentions that only USD is accepted as `quote`, but this isn't enforced in `getQuoteFromProvider()` either.

Recommendation: Consider refactoring `SuperOracle` to ensure consistent functionality:

- `provider` should represent an identifier of the oracle provider (e.g Chainlink or Redstone).
- The oracle should only return quotes for USD.

Superform: Fixed in PR 351 and PR 355.

Cantina Managed: Verified.

3.4 Informational

3.4.1 The code should check if the token address is `address(0)` and yield source is `address(0)`

Severity: Informational

Context: ApproveAndDeposit4626VaultHook.sol#L55

Description:

```
if (yieldSource == address(0) || account == address(0)) revert ADDRESS_NOT_VALID();
```

The code checks if account address is `address(0)`, however, the account address is `msg.sender` and cannot be `address(0)`.

Recommendation: In ApproveAndDeposit, the code should check if `yield source` or `token` is `address(0)`. In DepositHook and RedeemHook, the code should if `yield source` is `address(0)`.

Superform: Fixed in PR 312.

Cantina Managed: Verified, the recommended fix was implemented.

3.4.2 Minor improvements to code and comments

Severity: Informational

Context: *(See each case below)*

Description/Recommendation:

1. SuperExecutor.sol#L153-L159, SuperExecutor.sol#L166-L168 - Consider using the `_execute()` function for a single call. This should save some gas as there's no reason to perform a batch call.
2. HookDataDecoder.sol#L15 - The code can be simplified to:

   ```
   - return BytesLib.toAddress(BytesLib.slice(data, 4, 20), 0);
   + return BytesLib.toAddress(data, 4);
   ```

   Additionally, in ApproveERC20Hook.sol#L38-L41, hook data decoding can be simplified as such:

   ```
   - address token = BytesLib.toAddress(BytesLib.slice(data, 0, 20), 0);
   - address spender = BytesLib.toAddress(BytesLib.slice(data, 20, 20), 0);
   - uint256 amount = BytesLib.toUint256(BytesLib.slice(data, 40, 32), 0);
   + address token = BytesLib.toAddress(data, 0);
   + address spender = BytesLib.toAddress(data, 20);
   + uint256 amount = BytesLib.toUint256(data, 40);
   ```

   The same applies to `preExecute()` and `postExecute()`.

   In general, many places in the codebase unnecessarily call `BytesLib.slice()` before converting the sliced bytes to their actual type. Consider going through the codebase and simplifying all of these instances.
3. BaseLedger.sol#L20 - `SafeERC20` is unused and can be removed.
4. Swap1InchHook.sol#L26 - `aggregationRouter` can be declared immutable.
5. SuperNativePaymaster.sol#L94-L104 - Consider modifying the start and end offsets to be constants.
6. AcrossReceiveFundsAndExecuteGateway.sol#L56 - `superNativePaymaster` isn't used anywhere in the code and should be removed.
7. SuperMerkleValidator.sol#L169 - This check can be removed as `MerkleProof.verify()` is always checked later on in `_isSignatureValid()`.
8. SuperMerkleValidator.sol#L130 - Consider making `bytes4(0x1626ba7e)` a constant.
9. SuperNativePaymaster.sol#L144-L146 - Checking `mode == PostOpMode.postOpReverted` is redundant as `postOp()` will never be called with `PostOpMode.postOpReverted`, as documented in the `IPaymaster` interface. Additionally, EIP-4337 doesn't include `postOpReverted` in the `PostOpMode` enum.
10. SuperNativePaymaster.sol#L114-L116 - The `nodeOperatorPremium < 0` condition is redundant and can be removed as `nodeOperatorPremium` is a `uint128`.
11. SuperNativePaymaster.sol#L106-L108 - Consider using `unpackVerificationGasLimit()` and `unpackCallGasLimit()` from `UserOperationLib` instead of re-writing your own implementation in `PaymasterGasCalculator`.
12. BaseHook.sol#L40-L47 - `_decodeBool()` can be simplified to:

    ```
    function _decodeBool(bytes memory data, uint256 offset) internal pure returns (bool) {
        return data[offset] != 0;
    }
    ```

13. SuperOracle.sol#L196-L198 - This comment is incorrect as `_configureOracles()` is also called in the constructor.
14. SuperOracle.sol#L51-L55 - Setting each provider's staleness to `maxStaleness` in the constructor is not needed as `_configureOracles()` sets the provider's staleness to `maxStaleness` if it is currently `0`. Consider removing this code.
15. SuperOracle.sol#L47 - The `owner != address(0)` check is not needed and can be removed as `owner != address(0)` is already checked in `Ownable`.

16. BaseClaimRewardHook.sol#L12 - `obtainedReward` is never used and can be removed.
17. SwapOdosHook.sol#L31, ApproveAndSwapOdosHook.sol#L31 - `odosRouterV2` can be declared immutable.
18. ApproveAndSwapOdosHook.sol#L29, ApproveAndSwapOdosHook.sol#L29 - This comment is wrong, it should be:

```
- bool usePreviousHookAmount = _decodeBool(data, 168 + pathDefinition_paramLength + 20 + 4);
+ bool usePreviousHookAmount = _decodeBool(data, 188 + pathDefinition_paramLength + 20 + 4);
```

Superform: Fixed in PR 319 and PR 369.

Cantina Managed: Verified, the recommended fixes were implemented.

3.4.3 `validateSignatureWithData()` in validator modules should be removed

Severity: Informational

Context: SuperMerkleValidator.sol#L134-L154, SuperDestinationValidator.sol#L75-L94.

Description: In `SuperMerkleValidator` and `SuperDestinationValidator`, `validateSignatureWithData()` is currently not implemented according to ERC 7780. For example, the function does not validate the `hash` provided against the signature or account.

Recommendation: Consider removing `validateSignatureWithData()` if it is unused.

Superform: Fixed in PR 313 and PR 340.

Cantina Managed: Verified, the function has been removed in both validators.

3.4.4 `ApproveERC20Hook.postExecute()` should set `outAmount` to the current allowance

Severity: Informational

Context: ApproveERC20Hook.sol#L65-L71

Description: `ApproveERC20Hook.postExecute()` decodes the allowance amount from the provide hook data (i.e. `data` below) to determine the spender's allowance:

```
function postExecute(address prevHook, address, bytes memory data) external {
    if (_decodeBool(data, 72)) {
        outAmount = ISuperHookResult(prevHook).outAmount();
    } else {
        outAmount = BytesLib.toUint256(BytesLib.slice(data, 40, 32), 0);
    }
}
```

A better design would be to set `outAmount` as `IERC20.allowance(account, spender)` instead. If the call to `approve()` ever fails, using `allowance()` ensures `outAmount` is `0` instead of the intended allowance amount. This also provides a way for subsequent hooks to check if the approval was successful if they need to.

Recommendation: In `postExecute()`, consider setting `outAmount` to the spender's allowance as such:

```
function postExecute(address, address account, bytes memory data) external {
    address token = BytesLib.toAddress(data, 0);
    address spender = BytesLib.toAddress(data, 20);
    outAmount = IERC20(token).allowance(account, spender);
}
```

Superform: Fixed in PR 326.

Cantina Managed: Verified, the recommendation was implemented.

3.4.5 Additional safety check in `BaseLedger._updateAccounting()`

Severity: Informational

Context: BaseLedger.sol#L155-L158

Description/Recommendation: In `BaseLedger._updateAccounting()`, consider adding a `config.ledger == address(this)` check to ensure the `yieldSourceOracleId` used belongs to the current ledger contract:

```
  ISuperLedgerConfiguration.YieldSourceOracleConfig memory config =
      superLedgerConfiguration.getYieldSourceOracleConfig(yieldSourceOracleId);

  if (config.manager == address(0)) revert MANAGER_NOT_SET();
+ if (onfig.ledger != address(this)) revert INVALID_LEDGER_ADDRESS();
```

Note that it in the current implementation of `SuperExecutor`, it should be impossible for this check to fail.

Superform: Fixed in PR 330.

Cantina Managed: Verified, the recommended fix was implemented.

3.4.6 `Withdraw7540VaultHook` is incompatible with ERC 7540 vaults with fungible request IDs

Severity: Informational

Context: Withdraw7540VaultHook.sol#L108-L111

Description: In `Withdraw7540VaultHook`, `_getSharesBalance()` calls `claimableRedeemRequest()` with `requestId = 0` to determine the amount of claimable shares an `account` currently has:

```
function _getSharesBalance(address account, bytes memory data) private view returns (uint256) {
    address yieldSource = data.extractYieldSource();
    return IERC7540(yieldSource).claimableRedeemRequest(0, account);
}
```

ERC 7540 does permit vaults where `requestId` is always `0`:

> When `requestId == 0`, the Vault MUST use purely the `controller` to discriminate the request state. The Pending and Claimable state of multiple requests from the same `controller` would be aggregated. If a Vault returns `0` for the `requestId` of any request, it MUST return `0` for all requests.

Centrifuge's implementation is an example of this, and calling `claimableRedeemRequest()` with `requestId = 0` would work for their vault. However, if an ERC 7540 vault uses fungible request IDs, meaning that the `requestId` returned by `requestRedeem()` is not `0`, this implementation does not work since `claimableRedeemRequest(0, ...)` would always return `0` instead of the amount of claimable shares owned by the account. As a result, `Withdraw7540VaultHook` is not compatible with all ERC 7540 vaults permitted by the specification.

Recommendation: Consider documenting that `Withdraw7540VaultHook` is only compatible with ERC 7540 vaults where `requestId` is non-fungible. If there is a need to support vaults where `requestId` is non-zero in the future, a separate hook should be created.

Superform: Acknowledged, added a comment in PR 332 to document this issue.

Cantina Managed: Acknowledged.

3.4.7 `account` should not be overwritten in `AcrossTargetExecutor.handleV3AcrossMessage()`

Severity: Informational

Context: AcrossTargetExecutor.sol#L110-L122

Description: In `AcrossTargetExecutor.handleV3AcrossMessage()`, if the `account` address provided by the user has no code, a new account is deployed through `nexusFactory` and the `account` address is overwritten:

```
(bytes memory initData, bytes memory executorCalldata, bytes memory sigData, address account, uint256
↪  intentAmount) = abi.decode(message, (bytes, bytes, bytes, address, uint256));

// ...

// @dev we need to create the account
if (initData.length > 0 && account.code.length == 0) {
    (bytes memory factoryInitData, bytes32 salt) = abi.decode(initData, (bytes, bytes32));
    address computedAddress = nexusFactory.computeAccountAddress(factoryInitData, salt);
    account = nexusFactory.createAccount(factoryInitData, salt);
    if (account != computedAddress) revert INVALID_ACCOUNT();
}
```

However, could potentially be dangerous as it allows the `account` specified by the user to be different from the address of the account deployed by `nexusFactory`.

Recommendation: Consider checking that the deployed account has the same address as `account` instead:

```
  // @dev we need to create the account
  if (initData.length > 0 && account.code.length == 0) {
      (bytes memory factoryInitData, bytes32 salt) = abi.decode(initData, (bytes, bytes32));
-     address computedAddress = nexusFactory.computeAccountAddress(factoryInitData, salt);
-     account = nexusFactory.createAccount(factoryInitData, salt);
+     address computedAddress = nexusFactory.createAccount(factoryInitData, salt);
      if (account != computedAddress) revert INVALID_ACCOUNT();
  }
```

Superform: Fixed in PR 339.

Cantina Managed: Verified, the recommended fix was implemented.

3.4.8 `SwapOdsHook` should check if `sePrevHookAmount` flag is set

Severity: Informational

Context: SwapOdosHook.sol#L57-L65

Description: The code does not check if `sePrevHookAmount` flag is set in `SwapOdsHook.sol` and `ApproveAndSwapOdosHook.sol`. If the `sePrevHookAmount` flag is set to `true`, the input amount should be the previous hook's output amount:

```
address inputToken = BytesLib.toAddress(BytesLib.slice(data, 0, 20), 0);
uint256 inputAmount = BytesLib.toUint256(BytesLib.slice(data, 20, 32), 0);

executions = new Execution[](1);
executions[0] = Execution({
    target: address(odosRouterV2),
    value: inputToken == address(0) ? inputAmount : 0,
    callData: abi.encodeCall(
        IOdosRouterV2.swap, (_getSwapInfo(account, prevHook, data), pathDefinition, executor, referralCode)
    )
});
```

The native ETH balance inputAmount should match the inputAmount in `_getSwapInfo`. Because this native ETH balance is set before `_getSwapInfo`.

Recommendation:

```
  uint256 inputAmount = BytesLib.toUint256(BytesLib.slice(data, 20, 32), 0);

+ bool usePrevHookAmount = _decodeBool(data, 188 + pathDefinition_paramLength + 20 + 4);
+ if (usePrevHookAmount) {
+     inputAmount = ISuperHookResult(prevHook).outAmount();
+ }
```

Superform: Fixed in PR 346.

Cantina Managed: Verified, the recommended fix was implemented.