SUPERFORM

Superform V2 Core Security Review

Auditors

Sujith Somraaj, Security Researcher

Report prepared by: Sujith Somraaj

March 24, 2025

Contents

1 About Researcher

Sujith Somraaj is a distinguished security researcher and protocol engineer with over eight years of comprehensive experience in the Web3 ecosystem.

In addition to working as a Security researcher at Spearbit, Sujith is also the security researcher and advisor for bridge protocols, including LI.FI & Garden.Finance (over $16B in combined volume) and also is a former founding engineer at Superform, a yield aggregator with over $130M in TVL.

Sujith has experience working with protocols including Monad, Blast, ZkSync, LI.FI, Decent, Drips, SuperSushi Samurai, DistrictOne, Omni-X, Centrifuge, Tea.xyz, Paintswap, Bitcorn, Sweep n' Flip, Byzantine Finance, Variational Finance, Satsbridge and Angles

Learn more about Sujith on sujithsomraaj.xyz or on cantina.xyz

2 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of that given smart contract(s) or blockchain software. i.e., the evaluation result does not guarantee against a hack (or) the non existence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, I always recommend proceeding with several audits and a public bug bounty program to ensure the security of smart contract(s). Lastly, the security audit is not an investment advice.

This review is done independently by the reviewer and is not entitled to any of the security agencies the researcher worked / may work with.

3 Scope

src/core/**/*.sol

4 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

4.1 Impact

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.

4.2 Likelihood

High almost certain to happen, easy to perform, or not easy but highly incentivized

Medium only conditionally possible or incentivized, but still relatively likely

Low requires stars to align, or little-to-no incentive

4.3 Action required for severity levels

Critical Must fix as soon as possible (if already deployed)

High Must fix (before deployment if not already deployed)

Medium Should fix

Low Could fix

5 Executive Summary

Over the course of 14 days in total, Superform engaged with the researcher to audit the contracts described in section 3 of this document ("scope").

In this period of time a total of 56 issues were found.

Project Summary	
Project Name	Superform
Repository	superform-xyz/v2-contracts
Commit	2fec28ae5.....ebc06f39c0
Audit Timeline	Mar 3rd - Mar 20th
Methods	Manual Review
Documentation	Medium-Low
Testing Coverage	Medium-Low

Issues Found	
Critical Risk	3
High Risk	5
Medium Risk	7
Low Risk	14
Gas Optimizations	7
Informational	20
Total Issues	**56**

6 Findings

6.1 Critical Risk

6.1.1 Zero length proof is accepted as a valid proof if `root == leaf` in `SuperMerkleValidator`

Context: SuperMerkleValidator.sol#L221

Description: The `_isSignatureValid()` function does not correctly validate the relationship between the Merkle root and leaf nodes. An attacker can set the **merkleRoot** equal to the **leaf** value and provide an **empty proof array**, causing the `MerkleProof.verify()` function to return true without proper validation.

```
function _isSignatureValid(
    address signer,
    address sender,
    uint48 validUntil,
    bytes32 merkleRoot,
    bytes32 leaf,
    bytes32[] memory proof
)
    private
    view
    returns (bool)
{
    // This will always return true if merkleRoot == leaf (when proof length is zero)
    bool isValid = MerkleProof.verify(proof, merkleRoot, leaf);
    return isValid && signer == sender && validUntil >= block.timestamp;
}
```

Likelihood Explanation: Cost of attack is minimal. The user doesn't need to do anything; generate a leaf and pass it in the root without proof. So the likelihood is `HIGH`.

Impact Explanation: It completely bypasses the entire merkle validation and the validator can be bypassed regardless of the tree structure, so impact is `HIGH`

Recommendation: Consider adding validations to proof length to ensure only valid proofs are accepted

Superform: Fixed in PR-221 and PR-284

Researcher: Verified fix

6.1.2 Anyone can call `executeFeeSplitUpdate` function to set `feeSplit` values to zero

Context: PeripheryRegistry.sol#L84

Description: The function `executeFeeSplitUpdate` updates the `feeSplit` value after a one-week timelock period. However, due to insufficient sanity checks, anyone can call the function to set the `feeSplit` values to zero.

```
function executeFeeSplitUpdate() external {
    if (block.timestamp < feeSplitEffectiveTime) revert TIMELOCK_NOT_EXPIRED();  // <--
    ↪  the feeSplitEffectiveTime is zero after execution, and hence the condition will
    ↪  always pass
    feeSplit = proposedFeeSplit;
    proposedFeeSplit = 0;
    feeSplitEffectiveTime = 0;

    emit FeeSplitUpdated(feeSplit);
}
```

Recommendation: Consider adding a zero check to ensure the `executeFeeSplitUpdate` can only be called when there is a pending update in the queue.

```
function executeFeeSplitUpdate() external {
-   if (block.timestamp < feeSplitEffectiveTime) revert TIMELOCK_NOT_EXPIRED();
+  if (feeSplitEffectiveTime == 0 || block.timestamp < feeSplitEffectiveTime) revert
↪  TIMELOCK_NOT_EXPIRED();
    ....
}
```

Likelihood explanation: The likelihood is `HIGH` as anyone can call this anytime.

Impact explanation: The protocol will lose its fees and will never be able to monetize the product. Consider that the impact will be `HIGH`

Superform: Fixed in PR-215

Researcher: Verified fix

6.1.3 Improper encoding of `depositor` address, leads to permanent loss of user funds from refunds

Context: AcrossSendFundsAndExecuteOnDstHook.sol#L112

Description: Per across docs, the refunds are sent to the depositor address on the source chain.

However, in the following code, the depositor address used is not the user's smart account on the source chain; instead, it's the **acrossGatewayExecutor** contract address.

```
        executions[0] = Execution({
            target: spokePoolV3,
            value: acrossV3DepositAndExecuteData.value,
            callData: abi.encodeCall(
                IAcrossSpokePoolV3.depositV3Now,
                (
                    acrossGatewayExecutor, /// <--- depositor is acrossGatewayExecutor
                    ↪  (not the user's smart account)
                    acrossV3DepositAndExecuteData.recipient,
                    acrossV3DepositAndExecuteData.inputToken,
                    acrossV3DepositAndExecuteData.outputToken,
                    acrossV3DepositAndExecuteData.inputAmount,
                    acrossV3DepositAndExecuteData.outputAmount,
                    acrossV3DepositAndExecuteData.destinationChainId,
                    acrossV3DepositAndExecuteData.exclusiveRelayer,
                    acrossV3DepositAndExecuteData.fillDeadlineOffset,
                    acrossV3DepositAndExecuteData.exclusivityPeriod,
                    acrossV3DepositAndExecuteData.message
                )
            )
        });
    }
```

Therefore, if across triggers a refund due to the failure of user-op execution on the destination chain or for some other reason (improper message being sent), the funds are sent back to the **acrossGatewayExecutor** address, which is the `AcrossReceiveFundsAndExecuteGateway` contract.

Likelihood explanation: The failure of bridge transactions generally has a `LOW` likelihood. However, due to executing a source chain generated by user-op, the possibility of failing can be made `HIGH` by frontrunning.

Impact explanation: Permanent loss of user funds (`HIGH` impact). The contract has no recovery method/way to transfer out these excess funds.

Recommendation: To handle the refunds appropriately, Consider encoding the user's smart account as the `depositor` address.

Superform: Fixed in PR-206

Researcher: Verified fix

6.2 High Risk

6.2.1 Missing paymaster integration for cross-chain gas prefunding

Context: AcrossReceiveFundsAndExecuteGateway.sol#L129

Description: The `AcrossReceiveFundsAndExecuteGateway.sol` contract executes user operations on the destination chain via the EntryPoint contract. Still, it lacks a mechanism to utilize

the paymaster for gas prefunding when users have already paid for gas on the source chain.

Instead, it unconditionally forwards operations to the entry point, which will fail for all the destination transactions funded on the source chain.

```
IMinimalEntryPoint(entryPointAddress).handleOps(userOps, superBundler);
```

Recommendation: Implement conditional logic to determine whether the operation should use a paymaster for gas pre-funding.

Superform:

Researcher:

6.2.2 Withdrawal restriction after full entry consumption

Context: BaseLedger.sol#L169, ERC1155Ledger.sol#L87

Description: The `_processOutflow()` function in the `BaseLedger.sol` contract prevents users from withdrawing shares if all their ledger entries have been entirely consumed, even if they possess valid shares acquired through means other than Superform deposits.

In the `BaseLedger.sol` contract, the `_processOutflow()` function processes user withdrawals by consuming shares from their ledger entries. However, the function has a design constraint requiring users to record unconsumed shares in their ledger to process a withdrawal.

```
while (vars.remainingShares > 0) {
    if (vars.currentIndex >= vars.len) revert INSUFFICIENT_SHARES();

    LedgerEntry storage entry = ledger.entries[vars.currentIndex];
    uint256 availableShares = entry.amountSharesAvailableToConsume;

    if (availableShares == 0) {
        unchecked {
            ++vars.currentIndex;
        }
        continue;
    }

    // Process shares...
}
```

Suppose a user has ledger entries, but all entries have been fully consumed (i.e., amountSharesAvailableToConsume is 0 for all entries). In that case, the loop will increment `vars.currentIndex` until it exceeds `vars.len`, at which point the function reverts with `INSUFFICIENT_SHARES()`.

This design creates a critical limitation for users who may have legitimately acquired shares through means outside the Superform protocol (e.g., through secondary markets or transfers from other users). Such users will be unable to withdraw their shares through the protocol if:

- They previously had ledger entries that are now fully consumed

- They have not made any new deposits that would create fresh ledger entries

This prevents users from using the protocol until they make a new deposit, creating potential usability issues and limiting interoperability with external protocols.

Recommendation: Consider adding additional criteria to allow withdrawals from the above mentioned case:

```
// Example modification
function _processOutflow(
    address user,
    address yieldSource,
    uint256 amountAssets,
    uint256 usedShares,
    ISuperLedgerConfiguration.YieldSourceOracleConfig memory config
)
    internal
    virtual
    returns (uint256 feeAmount)
{
    Ledger storage ledger = userLedger[user][yieldSource];

    // If no ledger entries, allow withdrawal without fee
    if (ledger.entries.length == 0) return 0;

   // If all ledger entries are already consumed, allow withdrawal without fee
   if(vars.currentIndex == vars.len - 1 &&
    ↪  ledger.entries[vars.currentIndex].amountSharesAvailableToConsume == 0) return 0;

   // continue as follows
}
```

A similar issue exists in extending contracts like ERC1155Ledger.sol#L87, and a similar fix will be sufficient.

Superform: FIFO was removed here. This code doesn't exist anymore.

Researcher: Acknowledged

6.2.3 `SuperMerkleValidator` accepts signatures incompatible with ERC-4337 standard

Context: SuperMerkleValidator.sol#L156-L160

Description: The ERC-4337 standard requires that:

> The signature of the userOpHash must be validated, and upon a signature mismatch, it should return SIG_VALIDATION_FAILED (1) without reverting. Any other error must cause a revert.

Nevertheless, in SuperMerkleValidator, the signature neglects the `userOpHash`, enabling the user to sign encoded user data without incorporating the `block.chainId` and `entrypoint` addresses in the signature.

```
        leaf = _createLeaf(userOpData); <--- does not encode block.chainid /
        ↪  entrypoint in the merkle leaf

        /// the messageHash generation also has no block.chaindid / entrypoint details
        bytes32 messageHash = _createMessageHash(sigData.merkleRoot, leaf,
        ↪  userOpData.sender, userOpData.nonce);
        bytes32 ethSignedMessageHash =
        ↪  MessageHashUtils.toEthSignedMessageHash(messageHash);
```

This is dangerous as it opens the door to cross-chain replay attacks, especially if the smart account and validator share the same address across various chains (which is the case).

However, cross-entry point attacks are mitigated since the sender will typically only be the designated entry point address in most ERC 7579 wallet implementations. But if the smart wallet implementation has a configurable entry point / upgradeable entry point, such accounts are prone to replay attacks.

Likelihood Explanation: The issue has a `HIGH` likelihood as the signatures are valid indefinitely and have no chain ID encoded, so it is possible to replay signatures on any future deployments without issue.

Impact Explanation: The issue's impact is `HIGH` as it allows unauthorized transactions from the user's smart account, which can lead to possible stealing of funds in some cases.

Recommendation: Consider including the `userOpHash` as a part of the signature digest.

Superform: Partially fixed in PR-220. In our case, we are treating the merkle root as the **userOpHash**. This is a bit of a divergence from 4337 eip, but we are doing this to give this one signature experience.

Researcher: Acknowledged. The fixed included the **userOpHash** as a part of the signature but is not fully 4337 compliant and requires the user to sign a different hash than the **userOpHash**, which might make the validator incompatible with multiple integrations.

6.2.4 Signature expiry tampering in `SuperMerkleValidator`

Context: SuperMerkleValidator.sol#L19, SuperMerkleValidator.sol#L222

Description: The `SuperMerkleValidator.sol` contract allows attackers to modify the expiration timestamp (**validUntil**) of a signed user operation without invalidating the signature. This can lead to unauthorized transaction execution beyond the intended expiry time, effectively enabling signature replay attacks.

The contract's signature verification mechanism does not include the **validUntil** timestamp in the message that gets signed. The `_createMessageHash()` function only consists of the namespace, merkleRoot, leaf, sender address, and nonce:

```
function _createMessageHash(
    bytes32 merkleRoot,
    bytes32 leaf,
    address sender,
    uint256 nonce
)
    private
    pure
    returns (bytes32)
{
    return keccak256(abi.encode(namespace(), merkleRoot, leaf, sender, nonce));
}
```

While the contract checks `validUntil >= block.timestamp` during validation in the `_isSignatureValid()` function, the absence of **validUntil** in the signed message allows attackers to modify this parameter:

```
function _isSignatureValid(
    address signer,
    address sender,
    uint48 validUntil,
    bytes32 merkleRoot,
    bytes32 leaf,
    bytes32[] memory proof
)
    private
    view
    returns (bool)
{
    // Verify merkle proof
    bool isValid = MerkleProof.verify(proof, merkleRoot, leaf);
    return isValid && signer == sender && validUntil >= block.timestamp;
}
```

PoC:

```
function test_manipulatingSignatureDeadline() external {
        // valid amount
        uint256 amount = 1e18;

        // get tokens for deposit
        _getTokens(underlying, account, amount);

        // hooks
        address[] memory hooksAddresses = new address[](2);
        address approveHook = _getHookAddress(ETH, APPROVE_ERC20_HOOK_KEY);
        address depositHook = _getHookAddress(ETH, DEPOSIT_4626_VAULT_HOOK_KEY);
        hooksAddresses[0] = approveHook;
        hooksAddresses[1] = depositHook;
```

```
// hooks data
bytes[] memory hooksData = new bytes[](2);
bytes memory approveData = _createApproveHookData(underlying,
↪  yieldSourceAddress, amount, false);
bytes memory depositData = _createDeposit4626HookData(
    bytes4(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)), yieldSourceAddress,
    ↪  amount, false, false
);

hooksData[0] = approveData;
hooksData[1] = depositData;

uint256 sharesPreviewed = vaultInstance.previewDeposit(amount);

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

// merkle proof
//  -- leaf
bytes32 leaf = keccak256(
    abi.encodePacked(
        account, userOpData.userOp.nonce, userOpData.userOp.callData,
        ↪  userOpData.userOp.accountGasLimits
    )
);

//  -- proof
bytes32[] memory proof = new bytes32[](1);
proof[0] = keccak256(
    abi.encodePacked(
        account, userOpData.userOp.nonce, userOpData.userOp.callData,
        ↪  userOpData.userOp.accountGasLimits
    )
);

// -- root
bytes32 merkleRoot = proof[0];

{
    /// user signed this for 1 hours from now
    uint48 validUntil = uint48(block.timestamp + 1 hours);
    bytes32 messageHash =
        keccak256(abi.encode(validator.namespace(), merkleRoot, leaf, account,
        ↪  userOpData.userOp.nonce));
    bytes32 ethSignedMessageHash = keccak256(abi.encodePacked("\x19Ethereum
    ↪  Signed Message:\n32", messageHash));
```

```
            (uint8 v, bytes32 r, bytes32 s) = vm.sign(uint256(uint160(account)),
            ↪  ethSignedMessageHash);
            bytes memory signature = abi.encodePacked(r, s, v);

            validSigData = abi.encode(validUntil, merkleRoot, proof, signature);
            userOpData.userOp.signature = validSigData;

            /// warping time to 2 hours
            /// SHOULD NOT BE VALID
            vm.warp(block.timestamp + 2 hours);
            validSigData = abi.encode(block.timestamp + 2 hours, merkleRoot, proof,
            ↪  signature);

            ERC7579ValidatorBase.ValidationData result =
            ↪  validator.validateUserOp(userOpData.userOp, bytes32(0));
            uint256 rawResult = ERC7579ValidatorBase.ValidationData.unwrap(result);
            bool _sigFailed = (rawResult >> 255) & 1 == 1;
            uint48 _validUntil = uint48(rawResult >> 160);

            assertFalse(_sigFailed);
            console.log("validUntil signed", validUntil);
            console.log("validUntil now", _validUntil);
        }
    }
```

Impact Explanation: The impact of this issue is `HIGH` as this vulnerability could result in the execution of operations that were meant to expire, potentially leading to financial losses.

Likelihood Explanation: It has a very `LOW` likelihood of exploitation, as in real-time, the odds of bundler/block proposer withholding the transaction are not financially profitable in most-times.

Recommendation: Include the `validUntil` parameter in the message hash calculation:

```
function _createMessageHash(
    bytes32 merkleRoot,
    bytes32 leaf,
    address sender,
    uint256 nonce,
    uint48 validUntil  // Added validUntil parameter
)
    private
    pure
    returns (bytes32)
{
    return keccak256(abi.encode(namespace(), merkleRoot, leaf, sender, nonce,
    ↪  validUntil));
}
```

Update the _processSignature function to pass the validUntil value:

```
function _processSignature(
    SignatureData memory sigData,
    UserOpData memory userOpData
)
    private
    pure
    returns (address signer, bytes32 leaf)
{
    // Create leaf
    leaf = _createLeaf(userOpData);

    // Create message hash - now including validUntil
    bytes32 messageHash = _createMessageHash(
        sigData.merkleRoot,
        leaf,
        userOpData.sender,
        userOpData.nonce,
        sigData.validUntil  // Pass validUntil to be included in the hash
    );
    bytes32 ethSignedMessageHash =
    ↪  MessageHashUtils.toEthSignedMessageHash(messageHash);

    signer = ECDSA.recover(ethSignedMessageHash, sigData.signature);
}
```

This ensures that the `validUntil` timestamp is cryptographically bound to the signature, preventing tampering.

Superform: Fixed in PR-219

Researcher: Verified fix

6.2.5 Front-running `handleV3AcrossMessage` will force refunds on source chain

Context: AcrossReceiveFundsAndExecuteGateway.sol#L129

Description: The `AcrossReceiveFundsAndExecuteGateway.sol` contract receives funds and an arbitrary message from the Across bridge. When the funds arrive, it executes a user-signed action decoded from the received message.

However, since the signed action is already broadcast on the source chain, anyone front-running the entire process can use it directly at the entry point contract on the destination chain. One caveat is that this will work only if the user's smart account on the destination chain has enough funds for the transaction.

Consider the following scenario to visualize the attack:

- User A initiates a bridging action of 100 USDC and signs a user-op to deposit the money in the yearn vault on the destination chain.

- Since the signed user-op is already known from the transaction logs on the source chain, a malicious actor broadcasts it directly to the entry point contract on the destination chain.
- If the user already has 100 USDC, then the transaction succeeds.
- Now, when across bridge calls, `handleV3AcrossMessage`, the call reverts as the `handles` call will revert due to an invalid nonce, initiating a refund action on the source chain.

Likelihood Explanation: The issue is highly likely, as anyone can broadcast the action to an entry point on the destination chain with low attack costs.

Impact Explanation: The impact is `MEDIUM`. This can be used to permanently DoS the bridging action for certain targeted high-value transfers.

Recommendation: Consider try/catching the execution logic, and if the destination call fails, transfer the received funds from the bridge to the user's smart wallet and trigger the action again with a new nonce from the front end.

```
function handleV3AcrossMessage(
  address tokenSent,
  uint256 amount,
  address, //relayer; not used
  bytes memory message
) external {
  if (msg.sender != acrossSpokePool) revert INVALID_SENDER();
  ....
-  IMinimalEntryPoint(entryPointAddress).handleOps(userOps, superBundler);
+  try IMinimalEntryPoint(entryPointAddress).handleOps(userOps, superBundler) {
+    emit AcrossFundsReceivedAndExecuted(account);
+ } catch {
+   // no action, as funds are already transferred
+    emit AcrossFundsReceivedButExecutionFailed(account);
+  }
}
```

Superform: Fixed in PR-208

Researcher: Verified fix

6.3 Medium Risk

6.3.1 Functions `simulateHandleOp` and `simulateValidation` are permanently DoSed due to faulty implementation

Context: SuperNativePaymaster.sol#L71-L101

Description: The `SuperNativePaymaster.sol` contract implements two functions: `simulateHandleOp()` and `simulateValidation()` to help users with simulations using the EntryPointSimulations.sol contract.

However, these two functions will not function permanently as the contract calls `entrypoint` configured, which will not include these functions.

These function get the `entryPointWithSimulations` address from the `_getEntryPointWithSimulations()` function,

```
function _getEntryPointWithSimulations() private view returns (IEntryPointSimulations)
↪  {
        return IEntryPointSimulations(address(entryPoint));
    }
```

This function simply casts the configured entrypoint address, which will not have any simulation functions available and will be reverting permanently.

Recommendation: Remove the above-mentioned two functions as the simulations should not be available on-chain per the documentation of the contract EntryPointSimulations.sol#L40

Superform: Fixed in PR-255

Researcher: Verified fix

6.3.2 Privileged role and actions lead to centralization risks for users

Context: SuperRegistry.sol#L30, SuperOracle.sol#L121

Description: Several onlyOwner functions affect critical protocol state and semantics, and therefore, lead to centralization risk for users. Some examples are highlighted below:

- `setAddress()` can change critical addresses at any point to steal / permanently lock user funds.
- `setProviderMaxStaleness()` can set the staleness value to either a very large value / zero to DoS user operations.

Recommendation: Consider:

- Documenting the privileged role and actions for protocol user awareness.
- Enforcing role-based access control, where different privileged roles control different protocol aspects and are backed by different keys, to follow the separation-of-privileges security design principle.
- Privilege actions affecting critical protocol semantics should be locked behind timelocks so that users can decide to exit or engage.
- Following the strictest opsec guidelines for privileged keys, e.g., use of reasonable multi sig and hardware wallets.

Superform: Acknowledged, we'll do this.

Researcher: Acknowledged

6.3.3 Native Token Handling Failure in `SwapOdosHook`

Context: SwapOdosHook.sol#L88

Description: In the `SwapOdosHook` contract, the `_getBalance()` function is used to track token balances before and after swap execution. However, this function only supports ERC20 tokens and cannot handle native ETH:

```
function _getBalance(address account, bytes memory data) private view returns
↪  (uint256) {
    address outputToken = BytesLib.toAddress(BytesLib.slice(data, 72, 20), 0);
    return IERC20(outputToken).balanceOf(account);
}
```

The function attempts to call the ERC20 `balanceOf()` method on all token addresses, including when the output token is native ETH (typically represented by an address like 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE (or) address(0)).

Recommendation: Modify the _getBalance() function to handle the special case for native ETH:

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

    if (outputToken == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE || outputToken ==
    ↪  address(0)) {
        return account.balance;
    } else {
        return IERC20(outputToken).balanceOf(account);
    }
}
```

Superform: Fixed in PR-269

Researcher: Verified fix

6.3.4 Native token handling failure in `Swap1InchHook`

Context: Swap1InchHook.sol#L238

Description: The `Swap1InchHook.sol` contract includes validation logic that supports swaps involving native ETH as the destination token, but its balance tracking implementation fails to handle this special case.

The core issue is in the `_getBalance()` function:

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

    return IERC20(dstToken).balanceOf(dstReceiver);
}
```

This function is called in both `preExecute()` and `postExecute()` to measure the token balance

before and after the swap. For ERC20 tokens, this works correctly, but when **dstToken** is the native token address (0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE), the function attempts to call `balanceOf()` on an address that isn't a contract, causing the transaction to revert.

Recommendation: Modify the `_getBalance()` function to handle the special case for native tokens:

```
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;
    } else {
        return IERC20(dstToken).balanceOf(dstReceiver);
    }
}
```

Superform: Fixed in PR-268

Researcher: Verified fix

6.3.5 Multiple ways to bypass fees at the protocol level

Context: Global

Description: The Superform protocol charges users when they withdraw from it. In other words, if the hook is of type OUTFLOW, a fee is charged on the user's profits.

However, there are multiple ways to avoid these fees, which can affect the protocol's revenue. Some of the possible ways are:

- Only using Superform for redemptions. And app interfaces to deposit. In this case, there won't be any entries in the ledger, possibly making fee calculations to return `0` as the fee amount. This can happen vice-versa, where users can deposit via superform and redeem directly on the protocol to avoid paying super form fees.
- Using cloned hooks that return a different hook type. Users can deploy new hooks for the same functionality that won't charge fees.
- Using colluded oracle sources.

Recommendation: Consider enforcing multiple ways to monetize the protocol rather than just depending on the hook type and monetizing only on redemptions.

Superform: Acknowledged

Researcher: Acknowledged

6.3.6 Potential out-of-gas issues in `SuperExecutor` due to unbounded hook and executions limit

Context: SuperExecutor.sol#L86

Description: The `SuperExecutor.sol` contract contains unbounded loops and operations that could lead to excessive gas consumption or out-of-gas errors under certain conditions.

There are two primary concerns regarding gas limits in the contract:

1. In the _execute function, there's an unbounded iteration through hooks:

```
for (uint256 i; i < hooksLen;) {
    address currentHook = entry.hooksAddresses[i];
    _processHook(account, ISuperHook(entry.hooksAddresses[i]), prevHook,
    ↪  entry.hooksData[i]);
    prevHook = currentHook;
    // go to next hook
    unchecked {
        ++i;
    }
}
```

2. The _processHook function processes a potentially unlimited number of executions returned by the hook:

```
Execution[] memory executions = hook.build(prevHook, account, hookData);
// run hook execute
if (executions.length > 0) {
    _execute(account, executions);
}
```

These unbounded operations could lead to the following:

- Transactions reverting with "out of gas" errors if too many hooks are specified
- Users are unable to execute complex operations because the required gas exceeds block gas limits
- Potential denial of service (DoS) if a malicious hook intentionally returns a large number of executions

Recommendation: Consider bounding the number of hooks and executions returned by any hook to a meaningful limit to avoid hook hijacking (or) out-of-gas issues.

Superform: After further review, we decided to acknowledge this issue and not fix it. We believe it's up to the user to decide which hooks to use, including the size of executions in them, and they acknowledge this by signing over the transaction. Therefore, I believe we shouldn't introduce these checks

Researcher: Acknowledged

6.3.7 Possible MEV of deposits / withdrawals due to lack of slippage

Context: Deposit4626VaultHook.sol#L56, Withdraw4626VaultHook.sol#L67

Description: The `deposit()` and `redeem()` functions in 4626 are prone to price-inflation attacks, and no slippage checks are added to the default hooks to safeguard the user.

Since the bundler and mempool transactions are public, anyone can front-run a user transaction to reduce the share/asset output and extract value.

Per the ERC-4626 standard:

> If implementors intend to support EOA account access directly, they should consider adding a function call for deposit/mint/withdraw/redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits since they have no other means to revert the transaction if the exact output amount is not achieved.

Since the interactions here are through smart accounts, we can add the additional slippage checks to ensure that the expected output amount is achieved to avoid MEV.

Recommendation: To protect the user, Consider enforcing slippage protections at the hook level in the `postExecute`.

Superform: Acknowledged

Researcher: The issue will affect users who go through Superform; hence, adding slippage checks as a part of it is super necessary, as 4626 explicitly states that it's designed for non-EOA users.

6.4 Low Risk

6.4.1 Inconsistent default staleness handling

Context: SuperOracle.sol#L121

Description: There is an inconsistency in how the SuperOracle contract handles default staleness values. In the `_configureOracles()` function, a default staleness of 1 day is applied to providers with a staleness value of 0. However, in the `setProviderMaxStaleness()` function, no such validation exists, allowing an owner to explicitly set a provider's staleness to 0, which would effectively make oracle data always appear stale.

Recommendation: Modify the setProviderMaxStaleness function to be consistent with _configureOracles by preventing a staleness value of 0:

```
function setProviderMaxStaleness(uint256 provider, uint256 newMaxStaleness) external
↪  onlyOwner {
    // Prevent setting staleness to 0
    if (newMaxStaleness == 0) newMaxStaleness = 1 days;

    providerMaxStaleness[provider] = newMaxStaleness;
    emit ProviderMaxStalenessUpdated(provider, newMaxStaleness);
}
```

Superform: Fixed in PR-278

Researcher: Verified fix

6.4.2 Unbounded oracle staleness parameter

Context: SuperOracle.sol#L121

Description: The `SuperOracle.sol` contract allows setting arbitrary values for the maximum staleness period through the `setProviderMaxStaleness()` function without any upper bound validation. The staleness parameter determines how old price data can be before it's invalid. Setting this value too high could lead to accepting severely outdated and potentially manipulated price data.

```
// Current implementation with no upper bound
function setProviderMaxStaleness(uint256 provider, uint256 newMaxStaleness) external
↪  onlyOwner {
    providerMaxStaleness[provider] = newMaxStaleness;
    emit ProviderMaxStalenessUpdated(provider, newMaxStaleness);
}

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

Recommendation: Implement an upper bound for the staleness parameter to ensure price data freshness.

Superform: Fixed in PR-264

Researcher: Verified fix

6.4.3 No `MAX_PROVIDERS` validation in `_configureOracles()` and `setProviderMaxStaleness()` functions

Context: SuperOracle.sol#L32, SuperOracle.sol#L121

Description: The `SuperOracle.sol` contract fails to validate provider IDs against the **MAX_PROVIDERS** limit in functions configuring oracle providers. Provider IDs that exceed **MAX_PROVIDERS** (currently set to 10) can be configured but will never be accessible through the average calculation mechanism.

```
// Constant definition
uint256 private constant MAX_PROVIDERS = 10;

// Functions missing validation
function setProviderMaxStaleness(uint256 provider, uint256 newMaxStaleness) external
↪  onlyOwner {
    providerMaxStaleness[provider] = newMaxStaleness;
    emit ProviderMaxStalenessUpdated(provider, newMaxStaleness);
}

function _configureOracles(
    address[] memory bases,
    uint256[] memory providers,
    address[] memory oracleAddresses
) private {
    // No validation for provider IDs against MAX_PROVIDERS
}
```

This vulnerability allows:

- Configuration of oracle providers with IDs that exceed the MAX_PROVIDERS limit
- These providers will never be included in average price calculations
- Wasted storage and gas from configuring inaccessible providers

Recommendation: Add explicit validation for provider IDs in both `setProviderMaxStaleness()` and `_configureOracles()` functions

Superform: I added the check where the array was configured in PR-263 but didn't add it to `setProviderMaxStaleness`.

Researcher: Verified fix.

6.4.4 Inconsistent implementation between `getTVL()` and `getTVLByOwnerOfShares()` in `ERC7540YieldSourceOracle`

Context: ERC7540YieldSourceOracle.sol#L59-L79

Description: The implementation of the `getTVL()` and `getTVLByOwnerOfShares()` methods in the `ERC7540YieldSourceOracle.sol` contract is inconsistent. Although these functions aim to calculate vault value, they employ differing methods to acquire their share balances:

```
/// getTVLByOwnerOfShares uses:
uint256 shares = IERC7540(yieldSourceAddress).balanceOf(ownerOfShares);

/// getTVL uses:
address share = IERC7540(yieldSourceAddress).share();
uint256 totalShares = IERC20Metadata(share).totalSupply();
```

Recommendation: Consider using a consistent pattern to interact with the `share()` address

Superform: Fixed in PR-199

Researcher: Verified fix

6.4.5 Inconsistent parameter types for `yieldSourceOracleId`

Context: ISuperLedger.sol#L40

Description: The `yieldSourceOracleId` parameter is defined as **bytes4** in the `updateAccounting()` function but appears as **bytes32** in the `AccountingOutflowSkipped` event.

Recommendation: Standardize the type of yieldSourceOracleId across all interface definitions. If bytes4 is the intended type, update the event definition:

```
event AccountingOutflowSkipped(
    address indexed user,
    address indexed yieldSource,
    bytes4 indexed yieldSourceOracleId,
    uint256 amount
);
```

Superform: Fixed in PR-258

Researcher: Verified fix

6.4.6 Override and disable `deposit()` function from `BasePaymaster`

Context: SuperNativePaymaster.sol#L5

Description: The `SuperNativePaymaster.sol` contract inherits `BasePaymaster.sol` and overrides several selective methods. However, the `deposit()` function from `BasePaymaster` allows users to deposit to the Entrypoint without any ops.

Since this function is not overridden, it poses risks as the entire deposit will be refunded to the user at the end of the `handleOps()` function.

Recommendation: Consider disabling the `deposit()` function from `BasePaymaster` as only deposits via `handleOps()` function is considered safe due to immediate refunds in the same method.

Superform: Took BasePaymaster locally and removed that method in PR-277

Researcher: Verified fix

6.4.7 `SuperNativePaymaster` has no recovery mechanism for force-sent ETH or tokens

Context: SuperNativePaymaster.sol#L15

Description: The contract could receive ETH through force-send mechanisms, which it isn't designed for. Such ETH would remain stuck and unrecoverable. Force-sends are rare unless malicious effects arise, which were not found during the audit.

Another scenario is that users could accidentally send wrapped ETH in token form in hopes of increasing the Paymaster's deposit balance. This will likely happen in practice, and the underlying value will be lost.

Recommendation: The force-sent ETH could easily be handled using the `entryPoint.depositTo` call pass `address(this).balance` instead of `msg.value` for the value parameter.

To handle accidentally sent token scenarios, an additional `onlyOwner` protected function would be necessary to approve the owner or some other address to handle those tokens. This would only support ERC-20 based tokens and it would realistically be impossible to support every specification.

Superform: Fixed in PR-280

Researcher: Verified fix

6.4.8 Validate `yieldSourceOracleId` in `_updateAccounting()` function

Context: SuperExecutor.sol#L108

Description: The `_updateAccounting()` function in the `BaseLedger.sol` contract lacks proper validation to ensure the provided **yieldSourceOracleId** is the correct oracle for the given yieldSource. However, in the ledger contract, the **yieldSourceOracleId** is validated by checking if the manager address is not zero, and is inconsistent across the codebase.

Recommendation: Consider validating **yieldSourceOracleId** in `_updateAccounting()` function as follows:

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

+ if (config.manager == address(0)) revert MANAGER_NOT_SET();
```

Superform: Fixed in PR-250

Researcher: Verified fix

6.4.9 Unbounded fee extraction via malicious ledger

Context: SuperExecutor.sol#L133

Description: The `SuperExecutor.sol` contract places unlimited trust in ledger contracts to determine fee amounts, with no upper bounds or reasonableness checks. This design allows a malicious or compromised ledger to extract arbitrary tokens from user wallets during outflow operations, potentially draining their balance.

In the `_updateAccounting()` function of the `SuperExecutor.sol` contract, fee amounts are entirely determined by the ledger contract with no validation on the returned value:

```
// 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()
);

// If there's a fee to collect (only for outflows)
if (feeAmount > 0) {
    // Get the asset token from the hook
    address assetToken = ISuperHookResultOutflow(hook).asset();
    if (assetToken == address(0)) revert ADDRESS_NOT_VALID();
    if (IERC20(assetToken).balanceOf(account) < feeAmount) revert
    ↪  INSUFFICIENT_BALANCE_FOR_FEE();

    // Fee transfer execution
    Execution[] memory feeExecution = new Execution[](1);
    feeExecution[0] = Execution({
        target: assetToken,
        value: 0,
        callData: abi.encodeCall(IERC20.transfer, (config.feeRecipient, feeAmount))
    });
    _execute(account, feeExecution);
    // ...
}
```

The contract has two critical issues:

- **No Fee Caps:** There is no maximum limit on the fee amount a ledger can charge. A ledger could return the user's entire token balance as the "fee".
- **No Proportionality Check:** The code doesn't validate that the fee is reasonable in proportion to the transaction value. A ledger could return a fee amount that is 100x the actual transaction value.

This represents a significant security risk because:

- Ledger contracts are treated as fully trusted entities
- Users likely expect fees to be a small percentage of their transaction value
- The only upper bound is the user's balance (if (IERC20(assetToken).balanceOf(account) < feeAmount) revert INSUFFICIENT_BALANCE_FOR_FEE();)

Recommendation: Validate if the `feeAmount` is within the limits of the transaction as follows:

```
// 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()
);

+ if(feeAmount >  ISuperHookResult(address(hook)).outAmount()) revert INVALID_FEE();

// If there's a fee to collect (only for outflows)
if (feeAmount > 0) {
    // Get the asset token from the hook
    address assetToken = ISuperHookResultOutflow(hook).asset();
    if (assetToken == address(0)) revert ADDRESS_NOT_VALID();
    if (IERC20(assetToken).balanceOf(account) < feeAmount) revert
    ↪  INSUFFICIENT_BALANCE_FOR_FEE();

    // Fee transfer execution
    Execution[] memory feeExecution = new Execution[](1);
    feeExecution[0] = Execution({
        target: assetToken,
        value: 0,
        callData: abi.encodeCall(IERC20.transfer, (config.feeRecipient, feeAmount))
    });
    _execute(account, feeExecution);
    // ...
}
```

Superform: Fixed in PR-249

Researcher: Verified fix

6.4.10 Missing validation for inflow fee collection

Context: SuperExecutor.sol#L122

Description: The `SuperExecutor.sol` contract lacks explicit validation to ensure fees are only collected for outflow operations. While the code includes a comment indicating this intention, there's no programmatic enforcement, creating a potential vulnerability if ledger contracts return non-zero fee amounts for inflow operations.

In the `_updateAccounting()` function of the `SuperExecutor.sol` contract, there's a section that handles fee collection:

```
// 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()
);

// If there's a fee to collect (only for outflows)
if (feeAmount > 0) {
    // Get the asset token from the hook
    address assetToken = ISuperHookResultOutflow(hook).asset();
    if (assetToken == address(0)) revert ADDRESS_NOT_VALID();
    if (IERC20(assetToken).balanceOf(account) < feeAmount) revert
    ↪  INSUFFICIENT_BALANCE_FOR_FEE();

    // Fee collection logic follows...
}
```

The code includes a comment stating that fees should only be collected for outflows, but there's no validation to enforce this policy. If a ledger implementation incorrectly returns a non-zero fee amount for an inflow operation, the system would attempt to collect a fee, which contradicts the intended design.

This creates two risks:

- **Protocol Inconsistency**: The system may behave inconsistently with its documented behavior.
- **Unauthorized Fee Collection**: Users could be charged fees during inflow operations if a ledger implementation contains a bug or is malicious.

Recommendation: Add a validation check to ensure fees are only collected for outflow operations:

```
// If there's a fee to collect (only for outflows)
if (feeAmount > 0) {
    // Explicitly validate that this is an outflow operation
    if (_type != ISuperHook.HookType.OUTFLOW) revert FEES_NOT_ALLOWED_FOR_INFLOWS();

    // Continue with existing fee collection logic
    address assetToken = ISuperHookResultOutflow(hook).asset();
    // ...
}
```

Superform: Fixed in PR-248

Researcher: Verified fix

6.4.11 Floating pragma version specification

Context: Global

Description: Multiple contracts (or almost all contracts) under audit scope use a floating pragma version specification `pragma solidity >=0.8.28;`, which can lead to inconsistent compilation results, potential security vulnerabilities, and verification challenges.

This approach indicates that the contracts should be compiled with Solidity version 0.8.28 or any higher version. While this provides flexibility, it introduces several risks:

Inconsistent Compilation Results: Different compiler versions can produce different bytecode even for identical source code due to varying optimizations and implementation details.

Potential Security Issues: Future compiler versions may introduce changes in behavior or bugs that could affect the contracts in unpredictable ways.

Verification Challenges: The exact compiler version becomes ambiguous when verifying the contract on block explorers or during security audits.

Recommendation: Replace the floating pragma with a fixed version:

```
pragma solidity 0.8.28;
```

Superform: Fixed in PR-291

Researcher: Verified fix. Interfaces continue to use floating pragma to help integrators.

6.4.12 Missing zero price validation in `_processOutflow()` of `ERC1155Ledger`

Context: ERC1155Ledger.sol#L109

Description: In the `ERC1155Ledger.sol` contract, there is an inconsistency in how price per share validation is handled between inflow and outflow operations. The code explicitly checks for and rejects zero prices during inflows, but there is no equivalent validation during outflows.

In the `_updateAccounting()` function, when processing an inflow, the code correctly verifies that the price per share is not zero:

```
// In _updateAccounting for inflows
uint256 pps =
↪  IYieldSourceOracle(config.yieldSourceOracle).getPricePerShare(yieldSource);
if (pps == 0) revert INVALID_PRICE();
```

However, in the `_processOutflow()` function, the price per share is retrieved without validation:

```
// In _processOutflow
uint256 ppsNow =
↪  IYieldSourceOracle(config.yieldSourceOracle).getPricePerShare(yieldSource);
uint256 currentBasis = sharesConsumed * ppsNow / (10 ** ctx.decimals);
```

Recommendation: Add a validation check in the `_processOutflow()` function to ensure price consistency:

```
uint256 ppsNow =
↪  IYieldSourceOracle(config.yieldSourceOracle).getPricePerShare(yieldSource);
if (ppsNow == 0) revert INVALID_PRICE();
uint256 currentBasis = sharesConsumed * ppsNow / (10 ** ctx.decimals);
```

This brings the outflow logic in line with the inflow logic and prevents potential accounting errors or exploitation.

Superform: Fixed in PR-247

Researcher: Verified fix. The `ERC1155Ledger.sol` is now simplified and uses the pps from `BaseLedger.sol` and this checks are added there.

6.4.13 Gas cost escalation for frequent users during withdrawals due to unbounded ledger growth

Context: BaseLedger.sol#L168-L194

Description: The `_processOutflow()` function iterates through ledger entries sequentially to calculate cost basis. As the number of entries grows with usage, the gas cost for withdrawals increases linearly, disproportionately affecting frequent users.

Long-term users who make frequent deposits will face increasingly expensive withdrawal operations. Eventually, withdrawal costs may exceed the value of smaller positions, effectively creating a minimum economically viable withdrawal amount that increases over time. This could also lead to DoS in rare cases where the queue size exceeds a certain size.

```
while (vars.remainingShares > 0) {
    if (vars.currentIndex >= vars.len) revert INSUFFICIENT_SHARES();

    LedgerEntry storage entry = ledger.entries[vars.currentIndex];
    uint256 availableShares = entry.amountSharesAvailableToConsume;

    if (availableShares == 0) {
        unchecked {
            ++vars.currentIndex;
        }
        continue;
    }

    // ... processing logic ...
}
```

Recommendation: Optimize the withdrawal flow to reduce gas costs. Consider adding functionality to periodically compact the ledger after consumption (or) during deposits.

Superform: FIFO was removed here: https://github.com/superform-xyz/v2-contracts/pull/232

Researcher: Acknowledged, the fee calculation logic is entirely changed and this issue might not be relevant anymore.

6.4.14 Immutable manager assignment in `YieldSourceOracleConfig`

Context: SuperLedgerConfiguration.sol#L100

Description: The `SuperLedgerConfiguration.sol` contract permanently assigns the message sender as the yield source oracle configuration manager without providing any mechanism to transfer or update this management role, creating operational and security risks.

When a yield source oracle configuration is set via the _setYieldSourceOracleConfig function, the contract assigns msg.sender as the manager of that configuration:

```
yieldSourceOracleConfig[yieldSourceOracleId] = YieldSourceOracleConfig({
    yieldSourceOracle: yieldSourceOracle,
    feePercent: feePercent,
    feeRecipient: feeRecipient,
    manager: msg.sender,
    ledger: ledgerContract
});
```

Recommendation: Implement a manager transfer mechanism to allow current managers to transfer their role:

- Add a function for manager role transfers that can only be called by the current manager
- Optionally implement a two-step transfer process where the new manager must accept the role

Superform: Fixed in PR-267

Researcher: Verified fix

6.5 Gas Optimization

6.5.1 Unnecessary storage reads and duplicate storage accesses

Context: BaseLedger.sol#L142

Description: The `BaseLedger.sol` contract has multiple instances of redundant and inefficient storage access patterns, particularly in the `_processOutflow()` function. These unnecessary storage reads and duplicate accesses result in higher gas costs and reduced contract efficiency.

In the `_processOutflow()` function:

```
Ledger storage ledger = userLedger[user][yieldSource];

// But then still directly accessing storage instead of using the ledger reference:
vars.lastIndex = vars.currentIndex;
vars.lastSharesConsumed = sharesConsumed;
vars.remainingShares -= sharesConsumed;

// And again accessing storage directly:
userLedger[user][yieldSource].unconsumedEntries = vars.currentIndex;
```

In loop iterations, repeatedly accessing the same storage pointer:

```
while (vars.remainingShares > 0) {
    if (vars.currentIndex >= vars.len) revert INSUFFICIENT_SHARES();

    // Should use cached ledger reference consistently:
    LedgerEntry storage entry = ledger.entries[vars.currentIndex];
    // ...
}
```

Recommendation: Consistently use storage pointers: Initialize storage pointers at the beginning of functions and use them consistently throughout.

Superform: FIFO was refactored in this PR: https://github.com/superform-xyz/v2-contracts/pull/232. I don't think this issue is related anymore

Researcher: Acknowledged. The fee calculation is revamped entirely and this issue may not be relevant anymore.

6.5.2 Call into EntryPoint's receive fallback via `handleOps()` for gas savings

Context: SuperNativePaymaster.sol#L62

Description: The `handleOps()` function in `SuperNativePaymaster.sol` currently calls `entryPoint.depositTo...` directly, along with accompanying arguments, to complete a deposit call.

The `entryPoint` also appears to have implemented a receive fallback function from its StakeManager import, which would achieve the same effect but in a more gas-efficient manner.

Recommendation: Refactor the `handleOps()` to call into the receive fallback instead via a call that includes value to save a few hundred gas.

```
- entryPoint.depositTo{value : msg.value}(address(this));
+ payable(address(entryPoint)).call{value: msg.value}("");
```

Superform: Fixed in PR-273

Researcher: Verified fix

6.5.3 Make `superRegistry` variable immutable in `SuperRegistryImplementer`

Context: SuperRegistryImplementer.sol#L13

Description: The `superRegistry` variable is declared once within the constructor of the `SuperRegistryImplementer` contract. As this variable remains unchanged after its initialization, it lacks the `immutable` keyword, which could assist in optimizing gas usage.

Recommendation: Consider adding immutable keyword to the superRegistry variable as follows:

```
ISuperRegistry public immutable superRegistry;
```

Superform: Fixed in PR-251

Researcher: Verified fix

6.5.4 Missing constant for repeated `keccak256` hash

Context: SuperExecutor.sol#L103, BaseLedger.sol#L35

Description: The `keccak256("SUPER_LEDGER_CONFIGURATION_ID")` hash is calculated directly in the `_updateAccounting()` function instead of using a pre-computed constant value. Also, the `keccak256("SUPER_EXECUTOR_ID")` hash is calculated inside the `onlyExecutor` modifier.

This results in unnecessary gas consumption each time the function is called.

Recommendation: Declare the hash as a constant at the contract level in all possible contexts:

```
bytes32 private constant SUPER_LEDGER_CONFIGURATION_ID =
↪  keccak256("SUPER_LEDGER_CONFIGURATION_ID");

function _updateAccounting(address account, address hook, bytes memory hookData)
↪  private {
    ISuperHook.HookType _type = ISuperHookResult(hook).hookType();
    if (_type == ISuperHook.HookType.INFLOW || _type == ISuperHook.HookType.OUTFLOW) {
        ISuperLedgerConfiguration ledgerConfiguration =
            ISuperLedgerConfiguration(superRegistry.getAddress(SUPER_LEDGER_CONFIGURAT
            ↪  ION_ID));
       ....
    }
}
```

Superform: Fixed in PR-245

Researcher: Verified fix

6.5.5 Use custom errors instead of `require` to save gas

Context: BaseHook.sol#L49

Description: In `BaseHook.sol`, a `require` statement is used for validation instead of custom errors. This creates inconsistency across the codebase. Additionally, custom errors are more efficient since they are consistently encoded as bytes4 compared to string error messages.

Recommendation: Consider replacing the require statement with custom error as follows:

```
function _decodeBool(bytes memory data, uint256 offset) internal pure returns (bool) {
-    require(data.length >= offset + 1, "Data length insufficient");
+   if(data.length < offset + 1) {
+      revert INVALID_DATA_LEN();
+ }
}
```

Superform: Fixed in PR-218

Researcher: Verified fix

6.5.6 Function `_execute()` in `SuperExecutor.sol` could be optimized

Context: SuperExecutor.sol#L67-L80

Description: The `_execute()` function shows inconsistent local declarations and accesses the memory array even when a local variable has been defined, along with other potential gas optimizations.

Recommendation: Consider optimizing the `_execute()` function as follows:

```
function _execute(address account, ExecutorEntry memory entry) private {
    // execute each strategy
    address prevHook;
    address currentHook;
    uint256 hooksLen = entry.hooksAddresses.length;
    for (uint256 i; i < hooksLen; ++i) {
       currentHook = entry.hooksAddresses[i];
       _processHook(account, ISuperHook(currentHook), prevHook, entry.hooksData[i]);
       prevHook = currentHook;
     }
}
```

Superform: Fixed in PR-217

Researcher: Verified fix

6.5.7 Redundant address(0) check in `unregisterHook` function

Context: PeripheryRegistry.sol#L56

Description: The `registerHook()` function within `PeripheryRegistry` prevents address(0) from being registered as a valid hook. Therefore, there's no need to perform an address zero check in the `unregisterHook()` function, as this scenario is impossible.

Recommendation: Consider removing the redundant address(0) check in the `unregisterHook()` function.

```
function unregisterHook(address hook_) external onlyOwner {
   if (!isHookRegistered[hook_]) revert HOOK_NOT_REGISTERED();
-  if (hook_ == address(0)) revert INVALID_ADDRESS();
   ....
}
```

Superform: Fixed in PR-213

Researcher: Verified fix

6.6 Informational

6.6.1 Rename `ERC1155Ledger` to `ERC5115Ledger`

Context: ERC1155Ledger.sol#L11

Description: The `ERC1155Ledger` tracks 5115 vaults but is incorrectly named; it should be `ERC5115Ledger` to avoid confusion.

Recommendation: Consider renaming the contract to `ERC5115Ledger`

Superform: Fixed in 2be7af8706bbf70a634281a845cb52c317b237fb

Researcher: Verified fix

6.6.2 Implement pre-execution validations in all stake-hooks

Context: FluidStakeHook.sol#L50, FluidStakeWithPermitHook.sol#L50, FluidUnstakeHook.sol#L57, GearboxStakeHook.sol#L52, GearboxUnstakeHook.sol#L54

Description: All the above-mentioned `stake / unstake` hook contracts implements basic sanity checks of the `yieldSource` but fails to do validations on the **amount**, **deadline** or other relevant parameters

Recommendation: Implement basic sanity checks in the hooks:

```
uint256 amount = _decodeAmount(data);

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

Superform: Fixed in PR-262. Ignored `FluidStakeWithPermitHook` as it will be moved to the mocks folder.

Researcher: Verified fix

6.6.3 Validate `rewardToken` in `_getBalance()` function of `BaseClaimRewardHook.sol`

Context: BaseClaimRewardHook.sol#L23

Description: Since the BaseClaimRewardHook.sol is not expected to support native tokens as reward tokens, it'll be efficient to validate if the **rewardToken** is a valid ERC20 token address. By adding `address(0)` validations, we can check basic sanity.

Recommendation: Validate if `rewardToken` is `address(0)` in the `_getBalance()` function.

Superform: Fixed in PR-261

Researcher: Verified fix

6.6.4 Validate `rewardToken` is not `address(0)` in `YearnClaimOneRewardHook`

Context: YearnClaimOneRewardHook.sol#L37

Description: The `build()` function in `YearnClaimOneRewardHook.sol` validates if **yieldSource** is address(0) but does not validate if **rewardToken** is address(0), and is essential since the hook does not support native tokens.

Recommendation: Validate decoded **rewardToken** in `build()` function as follows:

```
    function build(
        address,
        address,
        bytes memory data
    )
        external
        pure
        override
        returns (Execution[] memory executions)
    {
        address yieldSource = BytesLib.toAddress(BytesLib.slice(data, 0, 20), 0);
        address rewardToken = BytesLib.toAddress(BytesLib.slice(data, 20, 20), 0);

        if (yieldSource == address(0) || rewardToken == address(0)) revert
        ↪  ADDRESS_NOT_VALID();

        .....
    }
```

Superform: Fixed in PR-260

Researcher: Verified fix

6.6.5 Missing `address(0)` checks in constructor of `BaseHook.sol`

Context: BaseHook.sol#L35

Description: The `BaseHook.sol` accepts two parameters **registry_** and **author_** in the constructor without validating that these addresses are not the zero address (address(0)). This omission could allow the contract to be deployed with invalid critical parameters.

Recommendation: Consider validating these two parameters as follows:

```
constructor(address registry_, address author_, ISuperHook.HookType hookType_)
↪  SuperRegistryImplementer(registry_) {
        if(author == address(0)) revert ZERO_ADDRESS_NOT_ALLOWED();
        author = author_;
        hookType = hookType_;
    }
```

Superform: Author is removed from the hooks here in PR-259

Researcher: As author is removed, this issue is considered fixed

6.6.6 No check for maximum node operator premium

Context: SuperNativePaymaster.sol#L39

Description: The `calculateRefund()` function in `SuperNativePaymaster.sol` allows for a node operator premium to be specified without any upper bound, which could lead to excessive fees.

Recommendation: Consider bounding this value to a reasonable limit.

```
uint256 constant MAX_NODE_OPERATOR_PREMIUM = 50;

// In _validatePaymasterUserOp function
if (nodeOperatorPremium > MAX_NODE_OPERATOR_PREMIUM) {
    revert EXCESSIVE_PREMIUM();
}
```

Superform: We will acknowledge this as this is set by Superbundler and we are unsure what is a reasonable max limit

Researcher: Acknowledged

6.6.7 Validate refunds before calling `withdrawTo()` function in `handleOps()`

Context: SuperNativePaymaster.sol#L64

Description: The `handleOps()` function unconditionally attempts to withdraw the entire deposit balance without first checking if there's anything to withdraw.

This will result in unnecessary gas consumption when there's no remaining balance to withdraw, as the contract will still make the external call.

Recommendation: Add a check to only execute the withdrawal if there's a positive balance:

```
function handleOps(PackedUserOperation[] calldata ops) public payable {
    if (msg.value == 0) {
        revert EMPTY_MESSAGE_VALUE();
    }
    entryPoint.depositTo{ value: msg.value }(address(this));
    entryPoint.handleOps(ops, payable(msg.sender));

    uint256 remainingDeposit = entryPoint.getDepositInfo(address(this)).deposit;
    if (remainingDeposit > 0) {
        entryPoint.withdrawTo(payable(msg.sender), remainingDeposit);
    }
}
```

Superform: Fixed in PR-256

Researcher: Verified fix

6.6.8 Add sanity check in constructor that `_entryPoint` is contract

Context: SuperNativePaymaster.sol#L24

Description: The setting of the entryPoint variable currently lacks any validation. It could be set to the null address or an EOA while the Paymaster contract gets successfully deployed.

Recommendation: Adding a sanity check for the constructor's `_entryPoint` parameter would address concerns by confirming it points to a contract. This can be achieved with a simple addition of:

```
require(address(_entryPoint).code.length > 0, "Passed _entryPoint is not currently a
↪  contract");
```

to ideally the BasePaymaster in the upstream dependency or within the Paymaster contract's constructor itself.

Additionally, OZ's Address library could be utilized, which implements an `isContract()` function which works similar to the above.

Superform: Fixed in PR-254

Researcher: Verified fix

6.6.9 Add validation checks in the constructor of `SuperRegistryImplementer`

Context: SuperRegistryImplementer.sol#L16

Description: The `SuperRegistryImplementer` abstract contract is designed to provide access to the superRegistry for child contracts. The constructor accepts an address parameter (superRegistry_) stored as an immutable state variable. However, the contract lacks basic sanity validation for this input.

Recommendation: Consider adding basic sanity checks to this variable as follows:

```
constructor(address superRegistry_) {
    if(superRegistry_ == address(0)) revert ZERO_ADDRESS();
    superRegistry = ISuperRegistry(superRegistry_);
}
```

Superform: Fixed in PR-252

Researcher: Verified fix

6.6.10 `SuperExecutor` is non ERC-7535 compliant

Context: SuperExecutor.sol#L126

Description: The current `SuperExecutor` assumes the vault asset to be ERC-20 compliant and enforces ERC-20-related balance checks. Hence, it will become incompatible with the native asset 4626 extension (ERC-7535).

Recommendation: Consider implementing native asset support in fee handling of `SuperExecutor` to support ERC-7535 vaults.

Superform: Fixed in PR-270

Researcher: Asset token can also be `0xeee.eee` in some cases, so good to support both `address(0)` and `0xeee.eee`

6.6.11 `SuperExecutor` is incompatible with fee-on-transfer and rebasing tokens

Context: SuperExecutor.sol#L137

Description: The `SuperExecutor.sol` contract implements strict balance verification after fee transfers that will cause transactions to fail when using fee-on-transfer or rebasing tokens.

The issue is found in the `_updateAccounting()` function:

```
// Get balance before transfer
uint256 balanceBefore = IERC20(assetToken).balanceOf(config.feeRecipient);

// Execute transfer
Execution[] memory feeExecution = new Execution[](1);
feeExecution[0] = Execution({
    target: assetToken,
    value: 0,
    callData: abi.encodeCall(IERC20.transfer, (config.feeRecipient, feeAmount))
});
_execute(account, feeExecution);

// Strict balance check
uint256 balanceAfter = IERC20(assetToken).balanceOf(config.feeRecipient);
if (balanceAfter - balanceBefore != feeAmount) revert FEE_NOT_TRANSFERRED();
```

This pattern fails for:

- **Fee-on-transfer tokens:** These tokens deduct a percentage fee on each transfer. When transferring feeAmount, the recipient will receive less than the full amount.
- **Rebasing tokens:** These tokens automatically adjust balances based on their protocol mechanics. Even during a single transaction, the effective balance might change due to rebasing.

Recommendation: If you wish to support these exotic ERC20 tokens as vault assets, consider using a percentage tolerance:

```
uint256 balanceAfter = IERC20(assetToken).balanceOf(config.feeRecipient);
// Allow for up to 5% fee (or whatever is deemed reasonable)
if (balanceAfter < balanceBefore + (feeAmount * 95 / 100)) revert
↪  FEE_NOT_TRANSFERRED();
```

Superform: Acknowledged

Researcher: Acknowledged

6.6.12 Declare max fee percent as constant

Context: SuperLedgerConfiguration.sol#L89, BaseLedger.sol#L204, ERC1155Ledger.sol#L130

Description: The `SuperLedgerConfiguration.sol` contract has a hardcoded maximum fee percentage of 10,000 (10_000), representing 100% of the yield, which can affect code readability and quality.

Recommendation: Consider declaring the value as a constant:

```
// Define a reasonable max fee percentage (e.g., 30%)
uint256 private constant MAX_FEE_PERCENT = 3_000;

function _setYieldSourceOracleConfig(
    bytes4 yieldSourceOracleId,
    address yieldSourceOracle,
    uint256 feePercent,
    address feeRecipient,
    address ledgerContract
)
    internal
    virtual
{
    // Other validations...

    if (feePercent > MAX_FEE_PERCENT) revert INVALID_FEE_PERCENT();

    // Rest of the function...
}
```

Superform: Fixed in PR-246

Researcher: Verified fix

6.6.13 Redundant condition check in `_processOutflow()` function

Context: BaseLedger.sol#L202, ERC1155Ledger.sol#L127

Description: In the `_processOutflow()` function of the BaseLedger contract, there is a redundant condition check:

```
if (profit > 0) {
    if (config.feePercent == 0) revert FEE_NOT_SET();

    feeAmount = (profit * config.feePercent) / 10_000;
}
```

The check if (config.feePercent == 0) revert FEE_NOT_SET(); is redundant because the same validation is already performed earlier in the `_updateAccounting()` function in all inheriting contracts.

```
// Only process outflow if feePercent is not set to 0
if (config.feePercent != 0) {
    feeAmount = _processOutflow(user, yieldSource, amountSharesOrAssets, usedShares,
    ↪  config);
    // ...
}
```

Recommendation: Remove the redundant condition check from the _processOutflow function.

Superform: Fixed in PR-232

Researcher: Verified fix

6.6.14 Duplicate NatSpec comment section in `SuperExecutor`

Context: SuperExecutor.sol#L24

Description: The SuperExecutor contract contains a duplicate NatSpec comment section. The comment section labeled `EXTERNAL METHODS` appears twice in the contract code, which may lead to confusion and indicates a lack of attention to code organization and documentation quality.

Recommendation: Consider removing the duplicative comment section (or) rename it accordingly to avoid confusion.

Superform: Fixed in PR-243

Researcher: Verified fix

6.6.15 Missing input validations in `execute` function of `SuperExecutor.sol`

Context: SuperExecutor.sol#L67

Description: The `SuperExecutor,sol` contract's **execute()** method accepts arbitrary calldata and directly decodes it into an ExecutorEntry structure without adequately validating its contents. This lack of proper input validation creates potential vectors for unexpected behavior (or) reverts without meaningful error message.

The contract blindly decodes the input data and passes it to the internal **_execute()** function without validating:

- That the array lengths match (entry.hooksAddresses.length == entry.hooksData.length)
- That hook addresses are not zero addresses
- That hook addresses are valid (maybe through interface checks)
- That hook data is properly formatted for each specific hook (not mandatory but would be a good defensive check to fail fast)

Recommendation: Implement basic input validation before executing any operations.

Superform: Fixed in PR-242

Researcher: Verified fix

6.6.16 Remove duplicate code documentation

Context: SuperRegistry.sol#L27

Description: A commented-out documentation tag `@inheritdoc ISuperRegistry` doesn't correspond to any implemented function, suggesting incomplete code in `SuperRegistry.sol`

```
mapping(bytes32 => mapping(address => bool)) private roles;

// ...

/*////////////////////////////////////////////////////////////
                            OWNER
////////////////////////////////////////////////////////////*/
/// @inheritdoc ISuperRegistry

/// @inheritdoc ISuperRegistry
function setAddress(bytes32 id_, address address_) external override onlyOwner {
    // ...
}
```

Recommendation: Consider removing the incomplete code documentation (or) implement the relevant function.

Superform: Fixed in PR-214

Researcher: Verified fix

6.6.17 Remove unused `roles` mapping

Context: SuperRegistry.sol#L18

Description: The `roles` mapping is declared in `SuperRegistry.sol` but never used anywhere in the contract, which wastes gas during deployment and could confuse future developers.

```
mapping(bytes32 => mapping(address => bool)) private roles;
```

Recommendation: Consider removing the unused roles mapping.

Superform: Fixed in PR-214

Researcher: Verified fix

6.6.18 Remove unchecked loop increments in `for` loops

Context: BaseHook.sol#L61, SuperOracle.sol#L50, SuperOracle.sol#L92, ERC5115YieldSourceOracle.sol#L88, AbstractYieldSourceOracle.sol#L76, AbstractYieldSourceOracle.sol#L106, AbstractYieldSourceOracle.sol#L110, AbstractYieldSourceOracle.sol#L123, AbstractYieldSourceOracle.sol#L216, AbstractYieldSourceOracle.sol#L267, AbstractYieldSourceOracle.sol#L273, AbstractYieldSourceOracle.sol#L305, SuperLedgerConfiguration.sol#L36, SuperLedgerConfiguration.sol#L67

Description: Solidity 0.8.22 introduces an overflow check optimization that automatically generates an unchecked arithmetic increment of the counter of for loops. Hence, explicitly doing it is unnecessary and will degrade code quality.

Recommendation: Remove unchecked loop increments as it serve no purpose.

Superform: Fixed in PR-212

Researcher: Verified fix

6.6.19 Function `getAddress()` can be `external`

Context: SuperRegistry.sol#L41

Description: The function `getAddress()` is declared **public** but is not used anywhere in the context. The public visibility identifier is reserved for functions that should be called both internally and externally. Hence, the function is only expected to be called externally; the function should be declared **external**.

Recommendation: Consider changing the visibility from **public** to **external**

Superform: Fixed in PR-216

Researcher: Verified fix

6.6.20 Remove unused code

Context: SuperRegistry.sol#L27, SuperRegistry.sol#L18

Description: The audit contracts have unused code, variable declarations, and redundant comments that can be removed.

Recommendation: Consider removing the unused code to improve code quality.

Superform: Fixed in PR-214

Researcher: Verified fix