Superform v2 Periphery Security Review

Auditors

Christoph Michel, Lead Security Researcher

MiloTruck, Lead Security Researcher

Noah Marconi, Lead Security Researcher

Ladboy233, Security Researcher

Ethan, Associate Security Researcher

Report prepared by: Lucas Goiriz

September 12, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Superform is a non-custodial yield marketplace. The protocol provides access to ERC-4626 vaults from any EVM chain.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Superform v2 Periphery according to the specific commit. Any modifications to the code will require a new security review.

3 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

3.1 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.

3.2 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

3.3 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

4 Executive Summary

Over the course of 20 days in total, Superform engaged with Spearbit to review the superform-v2-periphery protocol. In this period of time a total of **66** issues were found.

Summary

Project Name	Superform
Repository	superform-v2-periphery
Commit	1b34dc13
Type of Project	DeFi, Vaults
Audit Timeline	Jun 10th to Jun 30th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	0	0
High Risk	6	0	0
Medium Risk	14	0	0
Low Risk	19	0	0
Gas Optimizations	13	0	0
Informational	13	0	0
Total	**66**	**0**	**0**

5 Findings

5.1 Critical Risk

5.1.1 Malicious actor can overwrite other's user state via 1 wei vault share transfer to steal fund

Severity: Critical Risk

Context: SuperVault.sol#L509-L515

Description: When transferring `SuperVault` shares transfer, the code copies the entire state from the `from` account to `to` account.

```
// called like this in _update
ISuperVaultStrategy.SuperVaultState memory state = strategy.getSuperVaultState(from);
strategy.updateSuperVaultState(to, state);
```

The state contains redeem info and share price data.

```
struct SuperVaultState {
    uint256 pendingRedeemRequest; // Shares requested
    uint256 maxWithdraw; // Assets claimable after fulfillment
    uint256 averageRequestPPS; // Average PPS at the time of redeem request
    // Accumulators needed for fee calculation on redeem
    uint256 accumulatorShares;
    uint256 accumulatorCostBasis;
    uint256 averageWithdrawPrice; // Average price for claimable assets
}
```

Then the code allows the exploits below.

- Scenario 1:
    1. Alice create account 1 and account 2.
    2. Alice have a fulfilled claim request, Alice deposit some assets to get 1 unit of share in account 1.
    3. Alice transfer the share to account 2 to clone the fulfillment request and double withdraw.
- Scenario 2:
    1. Bob have a fulfillment state.
    2. Alice deposit some assets to get 1 unit of shares.
    3. Alice transfer the share to bob and bob's state get his entire state overwritten.

Both scenarios lead to loss of fund.

Recommendation: Consider separate the `SuperVaultState` into:

1. `SuperVaultSharesState`: The average cost basis for a user's total shares. The code can track the accumulatorCostBasis. To be able to correctly compute averages, the code also need to store the baseline (accumulatorShares).
2. `SuperVaultRedeemState`: A requestRedeem call can convert the ERC20 shares at the current user's cost basis into an state with a specific state that gets modified via cancel/fulfill/claim. (converted to pendingRedeemShares, pendingCostBasis).

Transferring shares should only affect the first state, updating the cost basis of the receiver as the average of their old position and the transferred shares at the avgCostBasisPerShare of `from`.

Reducing the `SuperVaultSharesState` of from by the transferred shares but keeping the cost basis per share the same.

5.2 High Risk

5.2.1 `strategistHooksRoot` permissions may be circumvented by any Strategist in favor of global permissions

Severity: High Risk

Context: SuperVaultAggregator.sol#L1009

Summary: The project team notes that vault creators/strategists who desires less permission for their vault may reduce the strategyProofs that their vault is allowed to execute hooks from via the `proposeStrategyHooksRoot` function in `SuperVaultAggregator`.

The intention is to differentiate between what hooks a particular vault strategist may call from the globally enabled list of hooks. Globally, the list of permitted hooks is not finalized, however, the project team noted a rough idea can be seen in the `registerHooks()` function in `BaseTest.t.sol` where the test hooks are registered for the test `SuperGovernor` instance:

```
function _registerHooks(Addresses[] memory A) internal returns (Addresses[] memory) {
    if (DEBUG) console2.log("---------------- REGISTERING HOOKS ----------------");
    for (uint256 i = 0; i < chainIds.length; ++i) {
        vm.selectFork(FORKS[chainIds[i]]);

        SuperGovernor superGovernor = SuperGovernor(_getContract(chainIds[i], SUPER_GOVERNOR_KEY));

        console2.log("Registering hooks for chain", chainIds[i]);
        if (DEBUG) {
            console2.log("deposit4626VaultHook", address(A[i].deposit4626VaultHook));
            console2.log("redeem4626VaultHook", address(A[i].redeem4626VaultHook));
            console2.log("approveAndRedeem4626VaultHook", address(A[i].approveAndRedeem4626VaultHook));
            console2.log("deposit5115VaultHook", address(A[i].deposit5115VaultHook));
            console2.log("redeem5115VaultHook", address(A[i].redeem5115VaultHook));
            console2.log("requestDeposit7540VaultHook", address(A[i].requestDeposit7540VaultHook));
            console2.log("requestRedeem7540VaultHook", address(A[i].requestRedeem7540VaultHook));
            console2.log("approveAndDeposit4626VaultHook",
            ↪  address(A[i].approveAndDeposit4626VaultHook));
            console2.log("approveAndDeposit5115VaultHook",
            ↪  address(A[i].approveAndDeposit5115VaultHook));
            console2.log("approveAndRedeem5115VaultHook", address(A[i].approveAndRedeem5115VaultHook));
            console2.log(
                "approveAndRequestDeposit7540VaultHook",
                ↪  address(A[i].approveAndRequestDeposit7540VaultHook)
            );
            console2.log("approveErc20Hook", address(A[i].approveErc20Hook));
            console2.log("transferErc20Hook", address(A[i].transferErc20Hook));
            console2.log("deposit7540VaultHook", address(A[i].deposit7540VaultHook));
            console2.log("withdraw7540VaultHook", address(A[i].withdraw7540VaultHook));
            console2.log("approveAndRedeem7540VaultHook", address(A[i].approveAndRedeem7540VaultHook));
            console2.log("swap1InchHook", address(A[i].swap1InchHook));
            console2.log("swapOdosHook", address(A[i].swapOdosHook));
            console2.log("approveAndSwapOdosHook", address(A[i].approveAndSwapOdosHook));
            console2.log("acrossSendFundsAndExecuteOnDstHook",
            ↪  address(A[i].acrossSendFundsAndExecuteOnDstHook));
            console2.log("fluidClaimRewardHook", address(A[i].fluidClaimRewardHook));
            console2.log("fluidStakeHook", address(A[i].fluidStakeHook));
            console2.log("approveAndFluidStakeHook", address(A[i].approveAndFluidStakeHook));
            console2.log("fluidUnstakeHook", address(A[i].fluidUnstakeHook));
            console2.log("gearboxClaimRewardHook", address(A[i].gearboxClaimRewardHook));
            console2.log("gearboxStakeHook", address(A[i].gearboxStakeHook));
            console2.log("approveAndGearboxStakeHook", address(A[i].approveAndGearboxStakeHook));
            console2.log("gearboxUnstakeHook", address(A[i].gearboxUnstakeHook));
            console2.log("yearnClaimOneRewardHook", address(A[i].yearnClaimOneRewardHook));
            console2.log("ethenaCooldownSharesHook", address(A[i].ethenaCooldownSharesHook));
```

```
            console2.log("ethenaUnstakeHook", address(A[i].ethenaUnstakeHook));
            console2.log("cancelDepositRequest7540Hook", address(A[i].cancelDepositRequest7540Hook));
            console2.log("cancelRedeemRequest7540Hook", address(A[i].cancelRedeemRequest7540Hook));
            console2.log("claimCancelDepositRequest7540Hook",
            ↪  address(A[i].claimCancelDepositRequest7540Hook));
            console2.log("claimCancelRedeemRequest7540Hook",
            ↪  address(A[i].claimCancelRedeemRequest7540Hook));
            console2.log("cancelRedeemHook", address(A[i].cancelRedeemHook));
        }

        // Register fulfillRequests hooks
        superGovernor.registerHook(address(A[i].deposit4626VaultHook), true);
        superGovernor.registerHook(address(A[i].redeem4626VaultHook), true);
        superGovernor.registerHook(address(A[i].approveAndRedeem4626VaultHook), true);
        superGovernor.registerHook(address(A[i].deposit5115VaultHook), true);
        superGovernor.registerHook(address(A[i].redeem5115VaultHook), true);
        superGovernor.registerHook(address(A[i].requestDeposit7540VaultHook), false);
        superGovernor.registerHook(address(A[i].requestRedeem7540VaultHook), false);

        // Register remaining hooks
        superGovernor.registerHook(address(A[i].approveAndDeposit4626VaultHook), true);
        superGovernor.registerHook(address(A[i].approveAndDeposit5115VaultHook), true);
        superGovernor.registerHook(address(A[i].approveAndRedeem5115VaultHook), true);
        superGovernor.registerHook(address(A[i].approveAndRequestDeposit7540VaultHook), true);
        superGovernor.registerHook(address(A[i].approveErc20Hook), false);
        superGovernor.registerHook(address(A[i].transferErc20Hook), false);
        superGovernor.registerHook(address(A[i].deposit7540VaultHook), true);
        superGovernor.registerHook(address(A[i].withdraw7540VaultHook), false);
        superGovernor.registerHook(address(A[i].approveAndRedeem7540VaultHook), true);
        superGovernor.registerHook(address(A[i].swap1InchHook), false);
        superGovernor.registerHook(address(A[i].swapOdosHook), false);
        superGovernor.registerHook(address(A[i].approveAndSwapOdosHook), false);
        superGovernor.registerHook(address(A[i].acrossSendFundsAndExecuteOnDstHook), false);
        superGovernor.registerHook(address(A[i].fluidClaimRewardHook), false);
        superGovernor.registerHook(address(A[i].fluidStakeHook), false);
        superGovernor.registerHook(address(A[i].approveAndFluidStakeHook), false);
        superGovernor.registerHook(address(A[i].fluidUnstakeHook), false);
        superGovernor.registerHook(address(A[i].gearboxClaimRewardHook), false);
        superGovernor.registerHook(address(A[i].gearboxStakeHook), false);
        superGovernor.registerHook(address(A[i].approveAndGearboxStakeHook), false);
        superGovernor.registerHook(address(A[i].gearboxUnstakeHook), false);
        superGovernor.registerHook(address(A[i].yearnClaimOneRewardHook), false);
        superGovernor.registerHook(address(A[i].cancelDepositRequest7540Hook), false);
        superGovernor.registerHook(address(A[i].cancelRedeemRequest7540Hook), false);
        superGovernor.registerHook(address(A[i].claimCancelDepositRequest7540Hook), false);
        superGovernor.registerHook(address(A[i].claimCancelRedeemRequest7540Hook), false);
        superGovernor.registerHook(address(A[i].cancelRedeemHook), false);
        // EXPERIMENTAL HOOKS FROM HERE ONWARDS
        superGovernor.registerHook(address(A[i].ethenaCooldownSharesHook), false);
        superGovernor.registerHook(address(A[i].ethenaUnstakeHook), true);
        superGovernor.registerHook(address(A[i].morphoBorrowHook), false);
        superGovernor.registerHook(address(A[i].morphoRepayHook), false);
        superGovernor.registerHook(address(A[i].morphoRepayAndWithdrawHook), false);
        superGovernor.registerHook(address(A[i].pendleRouterRedeemHook), false);
    }

    return A;
}
```

Description: A Strategist can opt out of verifying strategy level permissions and benefit from looser global permissions by passing in empty `strategyProof` arrays when calling `SuperVaultStgrategy.executeHooks`. Hook

validation occurs on `SuperVaultAggregator.validateHook` where a check occurs to see if both the global and strategy root vetoes have taken place:

```
bool globalHooksVetoed = _globalHooksRootVetoed;
bool strategyHooksVetoed = _strategyData[strategy].hooksRootVetoed;
if (globalHooksVetoed && strategyHooksVetoed) {
    return false;
}
```

If a strategy veto took place or no veto took place, the validation defers to the calldata arguments to verify that one of the two proofs exist:

```
if (lengthGlobalProof == 0 && lengthStrategyProof == 0) {
    return false;
}
```

In our exploit scenario, the calling strategist may pass in the global proof only and circumvent any strategy level restrictions:

```
// First try to verify against the global root if provided
if (lengthGlobalProof > 0 && !globalVetoed) {
    // Only validate against global root if it exists
    if (_globalHooksRoot != bytes32(0) && MerkleProof.verify(globalProof, _globalHooksRoot, leaf)) {
        return true;
    }
}
```

Not passing a strategy proof would be most simple but even passing a strategy proof that does NOT permit the `hookArgs` would still succeed as the global check returns before the strategy specific check.

Impact Explanation: Permissions the vault creator/main strategist believe they are enforcing are not enforced by the protocol. Global hooks such as ERC20 token transfers, when enabled globally, are available to every strategist for every vault.

Likelihood Explanation: The functionality is enabled as is and requires no unique circumstances to be exploitable. Any strategist may break out of their sandboxed permissions set by the `strategyProof`.

Recommendation: Reconsider what hooks are globally permitted. When a `strategistHooksRoot` exists, disallow global proofs and enforce strategy level permissions.

5.2.2 Controller and receiver cannot redeem shares after depositing fund to if receiver address differs from controller address

Severity: High Risk

Context: SuperVaultStrategy.sol#L108-L121

Description: the ERC20 shares are minted to `receiver` but `SuperVaultStrategy.state` is created (with relevant accumulatorShares and cost basis for shares) for `controller`.

```
strategy.handleOperation(msg.sender, receiver, assets, shares, ISuperVaultStrategy.Operation.Deposit);

_mint(receiver, shares);
```

Then both `receiver` and `controller` cannot redeem shares because.

- Transaction revert when the code substract `accumulatorShares` when the stragists fulfill `receiver` redeem request.

```
function _calculateCostBasis(
    SuperVaultState storage state,
    uint256 requestedShares
)
```

```
    private
    returns (uint256 costBasis)
{
    if (requestedShares > state.accumulatorShares) revert INSUFFICIENT_SHARES();

    // Calculate cost basis proportionally
    costBasis = requestedShares.mulDiv(state.accumulatorCostBasis, state.accumulatorShares,
    ↪  Math.Rounding.Floor);

    // Update user's accumulator state
    state.accumulatorShares -= requestedShares;
    state.accumulatorCostBasis -= costBasis;

    return costBasis;
}
```

- `controller` cannot create redeem request because `controller` has no shares to transfer to `escrow` address.

Recommendation: `controller` (msg.sender) should only pay for the assets but the `receiver` of both ERC20 shares and the corresponding state with the cost basis should be for `receiver`.

```
// use receiver
_handleDeposit(receiver, assets, shares);
```

5.2.3 Fee handling locks fees in `SuperVaultAggregator`

Severity: High Risk

Context: SuperVaultAggregator.sol#L904

Description: Upkeep fees are transferred into the `SuperVaultAggregator` using the `depositUpkeep` function. The function records a balance for the strategist `_strategistUpkeepBalance[strategist]`.

When `forwardPPS` is called, the balance may be reduced if the caller is not exempt. The strategist level accounting is updated but there is no transfer of funds out of the contract or a global accounting update to allow a fee recipient to pull the fees out of `SuperVaultAggregator`.

Recommendation: Update global accounting to record how many up tokens a fee recipient may claim, and access control a claiming function for them to call.

5.2.4 Lack of replay protection in PPS oracle

Severity: High Risk

Context: ECDSAPPSOracle.sol#L127-L147

Description: `ECDSAPPSOracle` signature schema does not have nonce and can be replayed once enough validator (more than `quorumRequirement`) signs the price update. The malicious user can push the outdated PPS price and break the share accounting in `SuperVault` and `SuperVaultStragety` via signature replay.

```
// Create message hash with all parameters- If anyare incorrect, the message hash will be different and
↪  the
// derived signer address will be incorrect- resulting in a revert
bytes32 messageHash =
    keccak256(abi.encodePacked(strategy, pps, ppsStdev, validatorSet, totalValidators, timestamp));
bytes32 ethSignedMessageHash = messageHash.toEthSignedMessageHash();
```

The signature schema also lacks `block.chainid` and `address(this)`, then if `strategy` address is deployed to two blockchain, the malicious user can push the outdated PPS price and break the share accounting in `SuperVault` and `SuperVaultStrategy` via cross-chain signature replay.

Recommendation:

```
- bytes32 messageHash =
    keccak256(abi.encodePacked(strategy, pps, ppsStdev, validatorSet, totalValidators, timestamp));
+ bytes32 messageHash =
    keccak256(abi.encodePacked(strategy, pps, ppsStdev, validatorSet, totalValidators, timestamp,
    ↪  nonces++));
```

Use `_domainSeparatorV4()` to include `block.chainid` and `address(this)` in signature schema.

5.2.5 Lack of `hookAddress` in merkle tree schema in `SuperVaultAggregator.sol`

Severity: High Risk

Context: SuperVaultAggregator.sol#L960-L966

Description: `_createLeaf()` double-hashes the calldata and ignores the hook address, so different hook addresses with identical encoded args share the same authorization leaf. An example is that the the `RequestDeposit7540VaultHook.sol` has the data format.

```
function _buildHookExecutions(
    address prevHook,
    address account,
    bytes calldata data
)
    internal
    view
    override
    returns (Execution[] memory executions)
{
    address yieldSource = data.extractYieldSource();
    uint256 amount = _decodeAmount(data);
    bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);

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

The `RequestRedeem7540VaultHook.sol` has the data format and

```
////////////////////////////////////////////////////////////*/
/// @inheritdoc BaseHook
function _buildHookExecutions(
    address prevHook,
    address account,
    bytes calldata data
)
    internal
    view
    override
    returns (Execution[] memory executions)
{
    address yieldSource = data.extractYieldSource();
    uint256 shares = _decodeAmount(data);
    bool usePrevHookAmount = _decodeBool(data, USE_PREV_HOOK_AMOUNT_POSITION);

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

Both hook follow the data format: `address + uint256` and both hooks have the same inspect function, which just packs the yield source address.

```
/// @inheritdoc ISuperHookInspector
function inspect(bytes calldata data) external pure returns (bytes memory) {
    return abi.encodePacked(data.extractYieldSource());
}
```

So a malicious caller can:

- Use `RequestDeposit7540VaultHook.sol` data to execute redeem action.
- Use `RequestRedeem7540VaultHook.sol` data to execute deposit action.

which leads to loss of funds.

Recommendation: The leaf should be created using hook `address + hookArgs`.

```
function _createLeaf(address hookAddress, bytes calldata hookArgs) internal pure returns (bytes32) {
    /// @dev note the leaf is just composed by the args, not by the address of the hook
    /// @dev this means hooks with different addresses but with the same type of encodings, will have
    ↪  the
    /// same authorization (same proof is going to be generated). Is this ok?
    return keccak256(bytes.concat(keccak256(abi.encode(hookAddress, hookArgs))));
}
```

5.2.6 Cancelled redeem requests make shares permanently unredeemable

Severity: High Risk

Context: SuperVaultStrategy.sol#L609, SuperVaultStrategy.sol#L612, SuperVaultStrategy.sol#L951

Summary: When a user cancels a pending redeem request, the deletion of the entire `superVaultState[controller]` struct removes properties that are required for subsequent redeem requests. When the user later tries to redeem their shares again, the fulfillment of the request will always fail.

Description: When a user initially deposits assets, the shares and the assets associated with the deposit are stored in `superVaultState[controller].accumulatorShares` and `superVaultState[controller].accumulatorCostBasis` respectively. These values can only be updated by way of SuperVaultStrategy's `_handleDeposit` function.

But if the user creates a redeem request and then cancels it, `_handleCancelRedeem` deletes the entire `superVaultState[controller]` struct before returning the shares back to the user from `SuperVaultEscrow`. This is a problem, because subsequent calls to `fulfillRedeemRequests` to process this user's shares will revert with the `INSUFFICIENT_SHARES()` error after checking if `requestedShares > state.accumulatorShares`. And even if a partial fix restored `accumulatorShares` alone, the zeroed-out `accumulatorCostBasis` would break the cost basis calculation, leading to incorrect fee and asset distribution.

Impact Explanation: The only way for a user to redeem their shares is by way of this asynchronous `requestRedeem` process. Users will be permanently unable to redeem their shares after having cancelled a previous request to redeem them.

Likelihood Explanation: This will happen for any cancelled redeem request, so it's as likely to occur as users are likely to cancel a request for any reason.

Recommendation: Consider storing a separate struct with the details of the pending request, so that `_handleCancelRedeem` can easily restore `superVaultState[controller].accumulatorShares`, `superVaultState[controller].accumulatorCostBasis`, and `superVaultState[controller].averageWithdrawPrice` with the values they held prior to the redeem request. This pending struct can be deleted when the request is fulfilled.

5.3 Medium Risk

5.3.1 `_toggleYieldSourceActivation` can DoSed via 1 wei transfer

Severity: Medium Risk

Context: SuperVaultStrategy.sol#L787-L802

Description: The code strictly requires TVL owner share is 0:

```
if (IYieldSourceOracle(yieldSource.oracle).getTVLByOwnerOfShares(source, address(this)) > 0) {
  revert INVALID_AMOUNT();
}
```

Then a user can send 1 share to this contract to DOS the `_toggleYieldSourceActivation`.

Recommendation: Consider remove the `getTVLByOwnerOfShares` check.

5.3.2 Anyone can reset the `emergencyWithdrawable` flag to false to block emergency action in `SuperVaultStrategy`

Severity: Medium Risk

Context: SuperVaultStrategy.sol#L814-L820

Description: `timestamp >= emergencyWithdrawableEffectiveTime = 0` is always true if there is no proposal, so anyone can reset `emergencyWithdrawable = false`.

Recommendation: Add the check:

```
if (emergencyWithdrawableEffectiveTime == 0) revert NoProposal();
```

5.3.3 VaultStrategy cannot be initialized because default `feeConfig` is used for validation

Severity: Medium Risk

Context: SuperVaultStrategy.sol#L85-L89

Description: The code check `feeConfig` instead of `feeConfig_`, which is always `false` and block the `strategy` initialization.

Recommendation: Check `feeConfig_` instead of `feeConfig` when the `strategy` is initializated.

5.3.4 `owner == controller` should be enforced when handling redemption

Severity: Medium Risk

Context: SuperVault.sol#L186-L187

Description: When redeeming, the `owner` transfer the shares `escrow`. However, the fulfillment request tries to reduce the controller's `accumulatorShares`, if the `controller` does not have shares, the transaction reverts.

Recommendation: Enforce `owner == controller` when redeeming.

5.3.5 Global vetoes are circumvented by strategist controlled calldata

Severity: Medium Risk

Context: SuperVaultAggregator.sol#L753-L755

Description: Hook validation returns false when both global and strategy level hooks are vetoed: `globalHooksVetoed && strategyHooksVetoed`. Later the internal `_validateSingleHook` validates against the global or strategy hook root depending on whether a veto exists and whether proof data has been provided.

In the case of a global veto, validation defers to the strategy. The project has clarified that in the event `global is vetoed we would want to return false regardless if local is vetoed or not`.

Recommendation: Update the veto checks to also return false when `globalHooksVetoed` alone is true.

5.3.6 Malicious strategist can bypass the slippage protection

Severity: Medium Risk

Context: SuperVaultStrategy.sol#L137-L160

Description: When hooks are executed in `executeHooks`, this parameter serves as slippage protection:

```
bool usePrevHookAmount = _decodeHookUsePrevHookAmount(hook, hookCalldata);
if (usePrevHookAmount && prevHook != address(0)) {
    vars.outAmount = _getPreviousHookOutAmount(prevHook);
    if (expectedAssetsOrSharesOut == 0) revert ZERO_EXPECTED_VALUE();
    uint256 minExpectedPrevOut = expectedAssetsOrSharesOut * (BPS_PRECISION - _getSlippageTolerance());
    if (vars.outAmount * BPS_PRECISION < minExpectedPrevOut) {
        revert MINIMUM_PREVIOUS_HOOK_OUT_AMOUNT_NOT_MET();
    }
}
```

A malicious strategist can supply arbitrary `args.expectedAssetsOrSharesOut` to disable the slippage protection and frontrun certain hook execution (swap execution) to extract funds. Same issue exists when the stragist executes `fulfillRedeemRequests` and `_processSingleFulfillHookExecution`. A malicious strategist can supply arbitrary `args.expectedAssetsOrSharesOut` to disable the slippage protection to make users receive too few assets.

```
if (vars.outAmount * BPS_PRECISION < expectedAssetOutput * (BPS_PRECISION - _getSlippageTolerance())) {
    revert MINIMUM_OUTPUT_AMOUNT_ASSETS_NOT_MET();
}
```

Recommendation: Charge deposit / collateral from `strategist` address and slash them offline if the `strategist` does execute transaction maliciously.

5.3.7 Unsafe Set Mutation During Iteration in `executeAddIncentiveTokens` and `executeRemoveIncentiveTokens`

Severity: Medium Risk

Context: SuperGovernor.sol#L772-L791

Description: In `executeAddIncentiveTokens`, removing `token` from the `_proposedWhitelistedIncentiveTokens` set while iterating over it causes the loop to terminate early, not every token in `_proposedWhitelistedIncentiveTokens` will be iterated over.

```
for (uint256 i; i < _proposedWhitelistedIncentiveTokens.length(); i++) {
    token = _proposedWhitelistedIncentiveTokens.at(i);

    _isWhitelistedIncentiveToken[token] = true;
    emit WhitelistedIncentiveTokensAdded(_proposedWhitelistedIncentiveTokens.values());

    // Remove from proposed whitelisted tokens
    _proposedWhitelistedIncentiveTokens.remove(token);
}
```

The same issue exists in `executeRemoveIncentiveTokens()`. The code remove `token` from `_proposedRemoveWhitelistedIncentiveTokens` inside the for loop.

Recommendation:

```
while (_proposedWhitelistedIncentiveTokens.length() != 0) {
    address token = _proposedWhitelistedIncentiveTokens.at(0);

    _isWhitelistedIncentiveToken[token] = true;
    _proposedWhitelistedIncentiveTokens.remove(token);
}
```

Adding test case to test `executeRemoveIncentiveToken` and `executeAddIncentiveTokens` is also highly recommended.

5.3.8 Emergency mechanism for replacing malicious strategists is undermined by pre-existing strategist proposals

Severity: Medium Risk

Context: SuperVaultAggregator.sol#L395-L417

Description: Malicious primary strategists have wide latitude to exploit Superform users. The main mitigation for dealing with this scenario is SuperGovernor's `changePrimaryStrategist` function, which allows the admin of SuperGovernor to swap the primary strategist of any strategy with a non-malicious account, bypassing the timelock and taking control of the strategy.

However, the normal mechanism for updating the primary strategist, a timelocked proposal submitted by one of the strategy's secondary strategists, takes precedence over this emergency mechanism. If a proposal already exists when the admin calls `changePrimaryStrategist`, it remains after the admin's change takes effect, and it can still be executed once the proposal's timelock expires.

A malicious primary strategist can exploit this by populating the vault's set of secondary strategists with accounts they control, and then frontrunning the SuperGovernor's attempt to rescue the vault with a proposal that returns control of the strategy to the malicious strategist.

The SuperGovernor admin can call `changePrimaryStrategist` again once the malicious proposal has been executed, but the attacker only needs to maintain control long enough to drain user funds, add another account they control as a secondary strategist, and use that secondary strategist account to propose another of their accounts as the new primary strategist. So they too can repeat the process, and will have a new opportunity to exploit users every time they regain control of the strategy.

Impact Explanation: Given the amount of damage a malicious strategist can do to users, and the lack of other mechanisms in the protocol to mitigate this potential damage, the impact of blocking the one clean recovery path for the strategy is substantial. The team also stated (after this issue was initially raised during the review) that this is a high-priority security concern.

Likelihood Explanation: The initial iteration of the exploit described above (frontrunning the first admin call to `changePrimaryStrategist`) does not require specialized knowledge to execute successfully. Repeating the process requires slightly more effort, but remains straightforward.

Recommendation: `changePrimaryStrategist` should clear existing strategist proposals. Additionally, consider clearing all secondary strategists from the strategy as well, since they could be controlled by the malicious strategist and will retain the power to propose new strategists.

5.3.9 Validators can fake the number of participants for a PPS transaction

Severity: Medium Risk

Context: ECDSAPPSOracle.sol#L174-L176, SuperVaultAggregator.sol#L875-L883

Description: `ECDSAPPSOracle._validateProofs()` does not check `validatorSet` and `totalValidators`, which represent the number of validators which signed for a PPS transaction and the total number of validators. As such, validators can collude and pass incorrect data for `validatorSet` and `totalValidators` to bypass the participation rate check in `SuperVaultAggregator._forwardPPS()`:

```
// C2.3) M/N Check: Check if enough validators participated
if (args.totalValidators > 0 && _strategyData[args.strategy].mnThreshold > 0) {
    // Calculate participation rate, scaled by 1e18
    uint256 participationRate = (args.validatorSet * 1e18) / args.totalValidators;
    if (participationRate < _strategyData[args.strategy].mnThreshold) {
        checksFailed = true;
        emit StrategyCheckFailed(args.strategy, "INSUFFICIENT_VALIDATOR_PARTICIPATION");
```

```
    }
}
```

The only requirement is that a sufficient number of validators must collude in order to pass quorum.

Recommendation: Add a `validatorSet == validSignatureCount` check in `ECDSAPPSOracle._validateProofs()`. Additionally, the total number of validators can be fetched from the number of validators registered in `SuperGovernor` (i.e. `_validatorsList.length`).

5.3.10 Hooks which exclude arguments in `inspect()` must not be whitelisted

Severity: Medium Risk

Context: SuperVaultAggregator.sol#L957-L965, Swap1InchHook.sol#L102-L106

Description: In `SuperVaultAggregator`, the hooks which strategists can execute are restricted by whitelisting the hook's arguments in a merkle tree:

```
/// @notice Creates a leaf node for Merkle verification from hook arguments
/// @param hookArgs The packed-encoded hook arguments (from solidityPack in JS)
/// @return leaf The leaf node hash
function _createLeaf(bytes calldata hookArgs) internal pure returns (bytes32) {
    /// @dev note the leaf is just composed by the args, not by the address of the hook
    /// @dev this means hooks with different addresses but with the same type of encodings, will have
    ↪  the
    /// same authorization (same proof is going to be generated). Is this ok?
    return keccak256(bytes.concat(keccak256(abi.encode(hookArgs))));
}
```

More specifically, arguments returned by the hook's `inspect()` function must be a leaf in the merkle tree for it to be executable.

However, there are hooks that must never be whitelisted because they have arguments that cannot be known ahead of time. Using `Swap1InchHook` as an example, consider a Uniswap V2/V3 swap. `inspect()` returns the receiver (`to`), input token (`token`) and pool address (`dex`):

```
if (selector == I1InchAggregationRouterV6.unoswapTo.selector) {
    (Address to, Address token,,, Address dex) =
        abi.decode(txData_[4:], (Address, Address, uint256, uint256, Address));
    packed = abi.encodePacked(to.get(), token.get(), dex.get());
} else if (selector == I1InchAggregationRouterV6.swap.selector) {
```

More importantly, the `amount` and `minReturn` parameters are excluded as it's not possible to hardcode the amount to swap and receive. If `Swap1InchHook` was whitelisted, a strategist could steal funds by performing a swap with `minReturn = 0` (i.e. no slippage) and sandwiching the swap transaction.

As such, not all Superform hooks are safe to be whitelisted for strategists to execute.

Recommendation: When whitelisting hooks, identify hooks which exclude arguments in `inspect()` and avoid whitelisting them. The current protections do not guarantee that a strategist cannot rug for all Superform hooks.

5.3.11 Multiple bundler IDs can point to the same address in `BundlerRegistry`

Severity: Medium Risk

Context: BundlerRegistry.sol#L12-L13

Description: `BundlerRegistry` stores bundler data in two mappings:

1. Each bundler ID points to an address.
2. Each address points to the data for a bundler.

```
mapping(address bundlerAddress => Bundler bundlerData) public bundlers;
mapping(uint256 bundlerId => address bundlerAddress) public bundlerIds;
```

However, this design makes it possible for two IDs to point to the same `bundlerAddress`, which is problematic as two IDs will point to the same data in the `bundlers` mapping. This has unintended side-effects, for example assume the following occurs:

- Call `registerBundler()` to register two bundlers. This returns two IDs:
  - `id_1` points to the address of `bundler1`.
  - `id_2` points to the address of `bundler2`.
- Call `updateBundlerAddress()` to update `id_2` to point to `bundler2`. Now both IDs point to the same address.
- Call `updateBundlerAddress()` to update `id_2` to point to `bundler1` again.

The data at `bundlers[bundler1]` will be deleted, causing `bundler1` to be unregistered even though `id_1` still points to it. The following Foundry test demonstrates the example above:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

import { Test } from "forge-std/Test.sol";
import { BundlerRegistry } from "src/periphery/BundlerRegistry.sol";

contract BundlerRegistryTest is Test {
    function test_twoBundlersSameAddress() public {
        // Deploy BundlerRegistry
        BundlerRegistry bundlerRegistry = new BundlerRegistry(address(this));

        // Bundler addresses
        address bundler1 = address(0x123);
        address bundler2 = address(0x456);

        // Register two bundlers
        bundlerRegistry.registerBundler(bundler1, "");
        bundlerRegistry.registerBundler(bundler2, "");

        // Get ID of bundler2
        (uint256 id_2,,,) = bundlerRegistry.bundlers(bundler2);

        // Update ID 2 to point to bundler1
        bundlerRegistry.updateBundlerAddress(id_2, bundler1);

        // bundler1 is registered
        assertTrue(bundlerRegistry.isBundlerRegistered(bundler1));

        // Update ID 2 to point to bundler2
        bundlerRegistry.updateBundlerAddress(id_2, bundler2);

        // bundler1 is no longer registered
        assertFalse(bundlerRegistry.isBundlerRegistered(bundler1));
    }
}
```

Recommendation: Consider storing bundler data with only one mapping. For example, if bundlers are identified by their address, `bundlerAddress` should be the identifier and bundler IDs should be removed.

5.3.12 Issues with `CREATE2` salt in `SuperVaultAggregator.createVault()`

Severity: Medium Risk

Context: SuperVaultAggregator.sol#L128-L137

Description: `SuperVaultAggregator.createVault()` deploys contracts with a `CREATE2` salt as the hash of their parameters and the current nonce:

```
// Create minimal proxies
superVault = VAULT_IMPLEMENTATION.cloneDeterministic(
    keccak256(abi.encodePacked(params.asset, params.name, params.symbol, currentNonce))
);
escrow = ESCROW_IMPLEMENTATION.cloneDeterministic(
    keccak256(abi.encodePacked(params.asset, params.name, params.symbol, currentNonce))
);
strategy = STRATEGY_IMPLEMENTATION.cloneDeterministic(
    keccak256(abi.encodePacked(params.asset, params.name, params.symbol, currentNonce))
);
```

However, there are two issues with this implementation:

1. Using `abi.encodePacked()` could cause the concatenation of `name` and `symbol` could collide. For example, `abi.encodePacked("AA", "B")` and `abi.encodePacked("A", "AB")` are the same.
2. Not including `msg.sender` in the salt allows different users to deploy vaults/escrows/strategies to the same address.

This is problematic if a user batches a transaction to `createVault()` with another transaction which interacts with the newly deployed contracts, since an attacker could front-run the vault deployment and change its parameters.

For example:

- Bob batches two transactions:
  - Calls `createVault()` to deploy a vault with `mainStrategist` as himself. This deploys a vault at address `0x123...`.
  - Calls `SuperVault.deposit()` to deposit into the vault at `0x123...`.
- Alice front-runs his transactions to call `createVault()` with the same parameters, except `mainStrategist` is her address.
- Alice's transaction deploys the vault at `0x123...`.
- When Bob's transactions are executed, he ends up depositing into the vault where Alice is the `mainStrategist`, instead of himself.

Recommendation:

1. Use `abi.encode()` to pack arguments.
2. Include `msg.sender` in the salt.

5.3.13 Incorrect rounding direction in `SuperVault.convertToAssets()`

Severity: Medium Risk

Context: SuperVault.sol#L329-L333

Description: According to the EIP-4626 specification, `convertToAssets()` should round down:

> convertToAssets. MUST round down towards 0.

However, `SuperVault.convertToAssets()` rounds up instead:

```
function convertToAssets(uint256 shares) public view override returns (uint256) {
    uint256 currentPPS = _getStoredPPS();
    if (currentPPS == 0) return shares;
```

```
    return Math.mulDiv(shares, currentPPS, PRECISION, Math.Rounding.Ceil);
}
```

Recommendation: Modify `convertToAssets()` to round down. Additionally, `previewMint()` must be modified to still round up, since it calls `convertToAssets()` currently.

5.3.14 Signatures with zero nonce cannot be invalidated in `SuperVault.invalidateNonce()`

Severity: Medium Risk

Context: SuperVault.sol#L295-L297

Description: In `SuperVault`, `invalidateNonce()` reverts if the `nonce` to invalidate is `bytes32(0)`:

```
function invalidateNonce(bytes32 nonce) external {
    if (nonce == bytes32(0) || _authorizations[msg.sender][nonce]) revert INVALID_NONCE();
    _authorizations[msg.sender][nonce] = true;
```

However, `authorizeOperator()` does not explicitly check for `nonce == bytes32(0)`, which means the zero nonce can be used for authorization signatures. As such, if a user generates a signature with a zero nonce and wants to cancel it later on, he will be unable to do so.

Recommendation: Remove the `nonce == bytes32(0)` check from `invalidateNonce()`:

```
  function invalidateNonce(bytes32 nonce) external {
-     if (nonce == bytes32(0) || _authorizations[msg.sender][nonce]) revert INVALID_NONCE();
+     if (_authorizations[msg.sender][nonce]) revert INVALID_NONCE();
      _authorizations[msg.sender][nonce] = true;
```

5.4 Low Risk

5.4.1 `type(IERC7540Operator).interfaceId` should be supported in `support interface` check

Severity: Low Risk

Context: SuperVault.sol#L468-L472

Description: According to ERC 7540 spec:

> All asynchronous Vaults MUST return the constant value true if either 0xe3bc4e65 (representing the operator methods that all ERC-7540 Vaults implement) or 0x2f0a18c5 (representing the ERC-7575 interface) is passed through the interfaceID argument.

However, the if the interface id is `type(IERC7540Operator).interfaceId` (`0xe3bc4e65`), the `supportInterface` check still return `false`.

Recommendation: Import `rc/vendor/standards/ERC7540/IERC7540Vault.sol` and return `type(IERC7540Operator).interfaceId`.

5.4.2 `maxMint` revert if pps oracle drops below 1.0

Severity: Low Risk

Context: SuperVault.sol#L341-L344

Description: If `pps` drops below 1.0, the `maxMint` function will revert as we compute "uint256.max / pps" which will overflow.

Recommendation: `maxMint` can return $2^{256} - 1$ if there is no limit on the maximum amount of shares that may be minted.

5.4.3 Escrow transfers should skip share price update

Severity: Low Risk

Context: SuperVault.sol#L509-L515

Description: When transferring the share to escrow, the code should skip the share price update and the escrow only takes shares as custody and cannot start a redemption request.

Recommendation:

```
-   if (from != address(0) && to != address(0)) {
+   if (from != address(0) && to != address(0) && to != address(escrow) && from != address(escrow)) {
      ISuperVaultStrategy.SuperVaultState memory state = strategy.getSuperVaultState(from);
      strategy.updateSuperVaultState(to, state);
  }
  super._update(from, to, value);
```

5.4.4 `SuperVaultStrategy` is not capable of handling native token transfer

Severity: Low Risk

Context: SuperVaultStrategy.sol#L437-L442

Description: When process hook execution, the caller can optionally specify the native ETH value.

```
(vars.success,) = vars.executions[j].target.call{ value: vars.executions[j].value
↪  }(vars.executions[j].callData);
```

But the `SuperVaultStragety` smart contract cannot receive native ETH and the function `executeHooks` is not marked as payable. Then the strategist cannot forward ETH when executing the hook.

Recommendation: Mark `executeHooks` as payble. Add `receive()` function to `SuperVaultStragety` smart contract should receive ETH after the hook execution.

```
receive() external payable {

}
```

5.4.5 No `SuperVaultAggregator.forwardPPS` validation on `updateAuthority` and `args.timestamp`

Severity: Low Risk

Context: SuperVaultAggregator.sol#L189

Description: The only validation on `updateAuthority` and `timestamp` are to determine if upkeep fee is required. This means:

- Out of order updates may be made (earlier timestamps to replace later ones).
- Anyone can call.

If the replay protection issue is addressed, this issue requires validator misbehavior or nonces that are not monotonically increasing.

Recommendation: Validate both parameters ensuring timestamp only increases and callers are among those expected.

5.4.6 `SuperVault` is not compliant with specification during paused state

Severity: Low Risk

Context: SuperVault.sol#L336-L344

Description: `maxMint` and `maxDeposit` do not account for paused state on the strategy (max is 0 while paused).

Recommendation: Return 0 when the strategy is paused.

5.4.7 Many ERC20 tokens not supported - fee on transfer, `cUSDCv3`, etc...

Severity: Low Risk

Context: SuperVault.sol#L104

Description: The protocol assumes a standard ERC20 without unusual behavior such as fee on transfer, transferring of amounts that differ from calldata arguments (`cUSDCv3`), etc... In the event one of these tokens are used, severe issues and loss of funds will occur.

Recommendation: Document token assumptions and consider whether token whitelisting is desirable. In the event these non standard tokens need to be supported, changes to the token transferring and accounting is needed. See weird-erc20 for more information.

5.4.8 Array length validation inconsistencies

Severity: Low Risk

Context: SuperVaultAggregator.sol#L209-L214, SuperVaultStrategy.sol#L141-L144, SuperVaultStrategy.sol#L174

Description: Many functions across the protocol accept arrays as parameters, and each function checks that the array length of each parameter is equal to all other array parameters before iterating through any of them. There are a few instances where one of the array parameters is not validated (or a validation is not needed):

- `executeHooks` (SuperVaultStrategy.sol#L137): omits `args.hookCalldata` validation.
- `batchForwardPPS` (SuperVaultAggregator.sol#L207): omits `args.totalValidators` validation.
- `fulfillRedeemRequests` (SuperVaultStrategy.sol#L163): unnecessarily checks that `args.controllers.length == controllersLength`. Since `controllersLength` is itself defined as `args.controllers.length`, this check is not needed.

Recommendation: The missing validations listed above should be added, and the unnecessary one should be removed.

5.4.9 Inconsistent exemption check between `forwardPPS` and `batchForwardPSS`

Severity: Low Risk

Context: SuperVaultAggregator.sol#L207-L230

Description: There is discrepancy checking the up fee can be Exempted when forward single PPS and batch forward PPS. When a single PPS is forwarded via `forwardPPS`, the function `_isExemptFromUpkeep` runs.

```
// Check if the update is exempt from paying upkeep
bool isExempt = _isExemptFromUpkeep(args.strategy, updateAuthority, args.timestamp);

// Create a new ForwardPPSArgs struct with updated isExempt and upkeepCost
_forwardPPS(
    ForwardPPSArgs({
        strategy: args.strategy,
        isExempt: isExempt,
        pps: args.pps,
        ppsStdev: args.ppsStdev,
        validatorSet: args.validatorSet,
        totalValidators: args.totalValidators,
        timestamp: args.timestamp,
        upkeepCost: SUPER_GOVERNOR.getUpkeepCostPerUpdate()
    })
);
```

The function `_isExemptFromUpkeep`:

- Check if upkeep payments are globally disabled in `SuperGovernor`.
- Check if Update is exempt if it is stale.
- Check if strategist is a `superform` strategist.
- Check if the `updateAuthority` is in the authorized callers list.

However, when `batchForwardPPS` is called, the code only check if upkeep payments are globally disabled in SuperGovernor, and `_isExemptFromUpkeep` is not called.

The checks below are missing when batch forward PPS:

- Check if Update is exempt if it is stale.
- Check if strategist is a superform strategist.
- Check if the updateAuthority is in the authorized callers list.

Recommendation: Call `_isExemptFromUpkeep` when batch forward PPS price.

5.4.10 Missing check for invalid `SuperAsset`

Severity: Low Risk

Context: SuperGovernor.sol#L262

Description: It is best practice (and the standard throughout the codebase) to validate that `address` parameters are not `address(0)`, but `setSuperAssetManager` does not check the value of the user-supplied `superAsset`.

Recommendation: Update the existing address validation to include the `superAsset` parameter:

```
if (_superAssetManager == address(0) || superAsset == address(0)) revert INVALID_ADDRESS();
```

5.4.11 `SuperVault` initialization and vault creation should revert if decimal query fails

Severity: Low Risk

Context: SuperVaultAggregator.sol#L153-L154, SuperVault.sol#L105

Description: When initializing `SuperVault` or calling `createVault` on SuperVaultAggregator, `asset.tryGetAssetDecimals` defaults to 18 if the `staticcall` to `token.decimals` does not succeed. It is unlikely that this call would fail, but it is plausible if, for example, the `decimals` getter on the token contract is implemented with a non-standard function signature. If this were to fail, and the real decimal value was any number other than 18, much of the accounting in the vault would be incorrect.

Recommendation: Both functions mentioned above (`SuperVault`'s `initialize` and SuperVaultAggregator's `createVault`) should revert if `!success`:

```
(bool success, uint8 assetDecimals) = params.asset.tryGetAssetDecimals();
if (!success) revert INVALID_DECIMALS();
```

5.4.12 Missing validation of `maxStaleness` parameter

Severity: Low Risk

Context: SuperGovernor.sol#L288, SuperGovernor.sol#L296, SuperGovernor.sol#L305-L310, SuperVaultAggregator.sol#L113

Description: The following functions set the `maxStaleness` property from a caller-supplied value without validating it first:

- `createVault` (SuperVaultAggregator.sol#L113).

- `setOracleMaxStaleness` (SuperGovernor.sol#L288).
- `setOracleFeedMaxStaleness` (SuperGovernor.sol#L296).
- `setOracleFeedMaxStalenessBatch` (SuperGovernor.sol#L305).

If the `maxStaleness` value is too low, every PPS update will be treated as stale. This will exempt the strategist from paying upkeep fees until `maxStaleness` is reset to an appropriate value.

Recommendation: The protocol should define a `MIN_STALENESS` value, and each of the above functions should require that `maxStaleness >= MIN_STALENESS`.

5.4.13 `BundlerRegistry` functions can be called on non-existent bundler IDs

Severity: Low Risk

Context: BundlerRegistry.sol

Description: The following functions do not check that the address returned by `bundlerIds[_bundlerId]` is not `address(0)`:

- `getBundler()`.
- `updateBundlerAddress()`.
- `updateBundlerExtraData()`.
- `updateBundlerStatus()`.

As such, it is possible to call these functions with a non-existent bundler ID, which ends up reading/writing to `bundlers[address(0)]`.

Recommendation: In the functions listed above, verify that `bundlerIds[_bundlerId] != address(0)`. Additionally, check that `_newAddress != address(0)` in `updateBundlerAddress()`.

5.4.14 `SuperVaultAggregator._validateSingleHook()` incorrectly assumes empty proofs are invalid

Severity: Low Risk

Context: SuperVaultAggregator.sol#L989-L995

Description: `SuperVaultAggregator._validateSingleHook()` assumes a hook is not in a merkle tree if the length of its proof is zero:

```
uint256 lengthGlobalProof = globalProof.length;
uint256 lengthStrategyProof = strategyProof.length;

// If both proofs are empty, the hook is not allowed
if (lengthGlobalProof == 0 && lengthStrategyProof == 0) {
    return false;
}
```

However, empty proofs are valid if a merkle root is the leaf itself (i.e. merkle tree with only one leaf node). For example, if a strategy only has one (hook address, arguments) whitelisted, the `strategyRoot` would simply be that leaf. However, the check shown above and the `lengthStrategyProof > 0` check later on reject empty proofs, so it wouldn't be possible to prove that the leaf exists.

Recommendation: Consider changing the parameters to:

```
bytes32[] calldata proof,
bool isGlobalProof
```

Instead of passing a global and strategy proof, pass one proofs array alongside a `bool` to determine if that proof is for the global or strategy root.

5.4.15 Fulfill hooks in `SuperGovernor` could be not in `_registeredHooks`

Severity: Low Risk

Context: SuperGovernor.sol#L37-L39

Description: `SuperGovernor` maintains two set of hooks, `_registeredHooks` and `_registeredFulfillRequestsHooks`:

```
// Hook registry
EnumerableSet.AddressSet private _registeredHooks;
EnumerableSet.AddressSet private _registeredFulfillRequestsHooks;
```

However, the current implementation is dangerous considering that `_registeredHooks` should always be a superset of `_registeredFulfillRequestsHooks`. `registerHook()` always reverts when adding a hook if it is already in the set. Similarly, `unregisterHook()` reverts when removing a hook if it isn't already in the set. This creates two issues.

1. If a hook is in `_registeredHooks` but not in `_registeredFulfillRequestsHooks`, trying to register it as a fulfill request hook reverts.
2. A hook can be in `_registeredFulfillRequestsHooks` but not `_registeredHooks` by calling `unregisterHook()` with `isFulfillRequestsHook_ = false` on a fulfill request hook. This violates the invariant that `_registeredFulfillRequestsHooks` is always a subset of `_registeredHooks`.

Recommendation: Consider just skipping emitting the event instead of reverting if a hook is/isn't in the set. For example:

```
function registerHook(address hook_, bool isFulfillRequestsHook_) external onlyRole(_GOVERNOR_ROLE) {
    if (hook_ == address(0)) revert INVALID_ADDRESS();

    if (isFulfillRequestsHook_ && _registeredFulfillRequestsHooks.add(hook_)) {
        emit FulfillRequestsHookRegistered(hook_);
    }
    if (_registeredHooks.add(hook_)) {
        emit HookApproved(hook_);
    }
}
```

Additionally, `unregisterHook()` doesn't need a `isFulfillRequestsHook_` parameter, it should just attempt to remove from both sets:

```
function unregisterHook(address hook_) external onlyRole(_GOVERNOR_ROLE) {
    if (_registeredFulfillRequestsHooks.remove(hook_)) {
        emit FulfillRequestsHookUnregistered(hook_);
    }
    if (_registeredHooks.remove(hook_)) {
        emit HookUnregistered(hook_);
    }
}
```

5.4.16 Merkle roots are not cleared when unregistering hooks in `SuperGovernor`

Severity: Low Risk

Context: SuperGovernor.sol#L402-L408

Description: In `SuperGovernor`, when `unregisterHook()` is called to unregister a hook, the data stored for that hook in `superBankHooksMerkleRoots` and `vaultBankHooksMerkleRoots` are not cleared.

This creates a potential footgun where a hook is unregistered while it has merkle root data. If the same hook is registered again later on, the merkle root data is unexpectedly preserved. This is even more problematic if a hook

has a pending merkle root update and is unregistered before the update is executed, since there is no way to cancel the update.

Recommendation: In `unregisterHook()`, delete the data in `superBankHooksMerkleRoots` and `vaultBankHooksMerkleRoots` for the hook being unregistered.

5.4.17 `EnumerableSet.AddressSet` can be used for lists in `SuperGovernor`

Severity: Low Risk

Context: SuperGovernor.sol#L54-L64

Description/Recommendation: `SuperGovernor` maintains a mapping and array of addresses for validators, relayers and executors:

```
// Validator registry
mapping(address validator => bool isValidator) private _isValidator;
address[] private _validatorsList;

// Relayer registry
mapping(address relayer => bool isRelayer) private _isRelayer;
address[] private _relayersList;

// Executor registry
mapping(address executor => bool isExecutor) private _isExecutor;
address[] private _executorsList;
```

Consider using `EnumerableSet.AddressSet` for these three lists as it makes the related functionality a lot simpler. Additionally, the functions for removing a validator/relayer/executor iterate through the entire list to find the corresponding entry. This could cost too much gas and revert if the list is extremely large, resulting in DOS (although unlikely to occur).

5.4.18 `ApproveAndGearboxStakeHook.inspect()` wrongly excludes the `token` address

Severity: Low Risk

Context: ApproveAndGearboxStakeHook.sol#L74-L76, ApproveAndFluidStakeHook.sol#L82-L87

Description: In `ApproveAndGearboxStakeHook`, the `inspect()` function does not include the address of the token being staked:

```
function inspect(bytes calldata data) external pure returns (bytes memory) {
    return abi.encodePacked(data.extractYieldSource());
}
```

In contrast, `ApproveAndFluidStakeHook` includes the `token` address in `inspect()`.

```
function inspect(bytes calldata data) external pure returns (bytes memory) {
    return abi.encodePacked(
        data.extractYieldSource(),
        BytesLib.toAddress(data, 24) //token
    );
}
```

As such, when validating executions involving `ApproveAndGearboxStakeHook`, the `token` address will not be validated and can be any arbitrary address.

Recommendation: Include the `token` address in the `inspect()` function of `ApproveAndGearboxStakeHook`.

5.4.19 `share` to `asset` ratio is 1:1 for `convertToShares` and `convertToAssets` when `currentPPS == 0` but not for `totalAssets`

Severity: Low Risk

Context: SuperVault.sol#L318

Description: There's a difference in handling of share `share` to `asset` ratio in the `convertToShares`/`convertToAssets` functions vs the `totalAssets` function:

- `convertToShares`: `if (currentPPS == 0) return assets.`
- `convertToAssets`: `if (currentPPS == 0) return shares.`
- `totalAssets`: `Math.mulDiv(supply, currentPPS, PRECISION, Math.Rounding.Floor)` (evaluates to 0 when `currentPPS` is 0).

Recommendation: Documenting the 0 handling behavior will assist integrators. Returning `supply` when `currentPPS == 0` would be consistent with `convertToShares`/`convertToAssets`. However, consider whether `convertToShares`/`convertToAssets` returning something other than 0 is desirable in the first place when `pps == 0` as 0 may indicate a larger problem.

5.5 Gas Optimization

5.5.1 Duplicate signer check can be more efficient

Severity: Gas Optimization

Context: ECDSAPPSOracle.sol#L162-L167

Description: When validating proofs, the code loops over the proof and extract signer address and looping over the valid signer count and validate there is no duplicate signer. The nested for loop is not gas efficient if there are a lot of signers.

Recommendation: Consider validating if the signer is sorted by keeping track `lastSigner` of to avoid nested for loop:

```
address lastSigner;

if (proofsLength == 0) revert ZERO_LENGTH_ARRAY();

// Process each proof
for (uint256 i; i < proofsLength; i++) {
    // Recover the signer from the proof
    address signer = ethSignedMessageHash.recover(proofs[i]);

    // Verify the signer is a registered validator
    if (!SUPER_GOVERNOR.isValidator(signer)) revert INVALID_VALIDATOR();

    // Check for duplicates or improper ordering
    if (signer <= lastSigner) revert INVALID_PROOF();
    lastSigner = signer;

    validSignatureCount++;
}
```

5.5.2 Use unchecked to save some gas

Severity: Gas Optimization

Context: SuperVaultAggregator.sol#L284

Description/Recommendation: A check occurs above confirming `_strategistUpkeepBalance[msg.sender] >= amount` meaning the subtraction can be safely unchecked `_strategistUpkeepBalance[msg.sender] -= amount`.

5.5.3 Save gas by avoiding duplicate hashing

Severity: Gas Optimization

Context: SuperVaultAggregator.sol#L129-L137

Description/Recommendation: The salt is the same for each of the 3 `cloneDeterministic` calls. Gas can be saved by performing the hash operation one time and reusing the value.

5.5.4 Avoid looping for lookups by using a `mapping` or `EnumerableSet`

Severity: Gas Optimization

Context: ISuperVaultAggregator.sol#L53

Description/Recommendation: When adding, removing, or checking for the presence of an account in the `authorizedCallers` array, a loop is used to inspect each element. Save gas, especially when checking for exemptions in `_isExemptFromUpkeep` by using a mapping or `EnumerableSet` to avoid loop with multiple `SLOAD`s.

5.5.5 Considering making `superGovernor` an immutable variable in the implementation contract

Severity: Gas Optimization

Context: SuperVaultStrategy.sol#L67

Description/Recommendation: `superGovernor` is not changed once initialized and would be more efficient as an immutable value in the implementation contract rather than initializing in storage on the proxy.

5.5.6 Save gas by skipping 0 value storage write

Severity: Gas Optimization

Context: SuperVaultAggregator.sol#L158

Description/Recommendation: Keeping the comment but skipping the 0 value storage write will save gas as the uninitialized value is already 0.

5.5.7 Do not duplicate address in storage and instead cast as IERC20 when needed

Severity: Gas Optimization

Context: SuperVaultEscrow.sol#L24-L26

Description/Recommendation: Having both `vault` and `shares` read from storage each time `escrowShares` or `returnShares` is called reads the same data from storage twice. Being in two separate storage slots also means both are cold reads resulting in unnecessary gas. Store only one of them and cast as needed. Casting an address as IERC20 incurs no gas cost.

5.5.8 `EnumerableSet` already performs a existence check in the `remove` function

Severity: Gas Optimization

Context: SuperVaultAggregator.sol#L405-L408, SuperVaultAggregator.sol#L450-L452

Description: `remove` already performs the check and simply returns false if address is not present. See snippet below:

```
// We cache the value's position to prevent multiple reads from the same storage slot
uint256 position = set._positions[value];

if (position != 0) {
    // ...snip...

    return true;
```

```
} else {
    return false;
}
```

Modifying to remove the conditional will save some gas:

```
- if (_strategyData[strategy].secondaryStrategists.contains(newStrategist)) {
    _strategyData[strategy].secondaryStrategists.remove(newStrategist);
- }
```

5.5.9 Save gas by caching values instead of re-reading from storage

Severity: Gas Optimization

Context: SuperVaultAggregator.sol#L493-L494, SuperVaultAggregator.sol#L514, SuperVaultAggregator.sol#L546-L548, SuperVaultAggregator.sol#L571

Description/Recommendation: Events emitting newly stored storage values can save gas by caching the value before writing and using to both write to storage and emit the event.

5.5.10 Parameter should be validated early in function logic

Severity: Gas Optimization

Context: SuperVaultStrategy.sol#L506

Description/Recommendation: The validation of `expectedAssetOutput` can be moved to the top of `_processSingleFulfillHookExecution` (or even earlier in the `fulfillRedeemRequests` loop, alongside the hook validation) for some gas savings in the event that `expectedAssetOutput == 0`.

5.5.11 Use `strategiesLength` variable to validate the lengths of the other parameters

Severity: Gas Optimization

Context: SuperVaultAggregator.sol#L216

Description/Recommendation: `args.strategies.length` is used to validate the lengths of all other arrays in `args`, but the variable `strategiesLength` is only declared after these four checks occur. Moving the declaration of `strategiesLength` above these validations, and using it instead of reading the length each time, will reduce the gas cost of the function.

5.5.12 Duplicated checks can be removed

Severity: Gas Optimization

Context: SuperGovernor.sol#L204, SuperVaultAggregator.sol#L401, SuperVault.sol#L196-L197, SuperVaultStrategy.sol#L949-L950

Description/Recommendation: In two instances, there are duplicated validations across tightly coupled functions:

- SuperGovernor's `changePrimaryStrategist` function validates that `newStrategist` is not `address(0)` before passing it onto SuperVaultAggregator, which implements the same check.
- SuperVault's `cancelRedeem` function validates that the quantity of shares in the controller's pending redeem request is not 0 before passing it onto SuperVaultStrategy, which implements the same check.

In both cases, the second function called in the chain can only be accessed by these user-facing entry points. The first instance of each check should be removed, leaving the validation closest to the critical logic intact (in case additional entry points are ever added in the future).

5.5.13 Duplicate `WhitelistedIncentiveTokensRemoved` event emission inside for loop

Severity: Gas Optimization

Context: SuperGovernor.sol#L810-L826

Description:

```
for (uint256 i; i < _proposedRemoveWhitelistedIncentiveTokenslength(); i++) {
    token = _proposedRemoveWhitelistedIncentiveTokens.at(i);
    if (_isWhitelistedIncentiveToken[token]) {
        _isWhitelistedIncentiveToken[token] = false;

        emit WhitelistedIncentiveTokensRemoved(_proposedWhitelistedIncentiveTokens.values());
    }
    // Remove from proposed whitelisted tokens to be removed
    _proposedRemoveWhitelistedIncentiveTokens.remove(token);
}
```

Note that the event `WhitelistedIncentiveTokensRemoved` emits inside the loop, which is can be gas consuming. The same (shrinking) array is broadcast many times.

Recommendation:

```
+ emit WhitelistedIncentiveTokensRemoved(_proposedRemoveWhitelistedIncentiveTokens.values());

  for (uint256 i; i < _proposedRemoveWhitelistedIncentiveTokens.length(); i++) {
      token = _proposedRemoveWhitelistedIncentiveTokens.at(i);
      if (_isWhitelistedIncentiveToken[token]) {
          _isWhitelistedIncentiveToken[token] = false;

-         emit WhitelistedIncentiveTokensRemoved(_proposedWhitelistedIncentiveTokens.values());
      }
      // Remove from proposed whitelisted tokens to be removed
      _proposedRemoveWhitelistedIncentiveTokens.remove(token);
  }
```

5.6 Informational

5.6.1 `mintShares` function is not used and can be removed

Severity: Informational

Context: SuperVault.sol#L437-L440

Description: `mintShares` is not used, the share is direclty when `deposit / mint` in the vault.

Recommendation: Remove the function `mintShares` to save gas.

5.6.2 Add reentrancy guard to hook execution

Severity: Informational

Context: Bank.sol#L34

Description/Recommendation: `SuperBank` and `VaultBank` rely on the hooks to prevent reentrancy whereas `SuperVaultStrategy` goes further and adds a reentrancy guard to `executeHooks`. Recommended adding the same `nonReentrant` protection as `superform/core` is undergoing modifications and may not always be safe.

5.6.3 `UpDistributor.owner` trust assumptions

Severity: Informational

Context: UpDistributor.sol#L43

Description/Recommendation: Owner can increase/decrease claimable amounts. If increased and after a user has already claimed, the user cannot claim a second time (bool for tracking claimed vs uint256 to track amount claimed). If decreased prior to claiming, the user misses out on tokens they would have otherwise been able to claim.

5.6.4 Strategists may circumvent upkeep fees

Severity: Informational

Context: SuperVaultAggregator.sol#L947-L952

Description: `SuperVaultAggregator` tracks an `authorizedCallers` array for each strategy. Any strategist may add accounts to this array: `if (!isAnyStrategist(msg.sender, strategy)) revert UNAUTHORIZED_UPDATE_AUTHORITY()`. When checking for exemption from upkeep fees in `SuperVaultAggregator._isExemptFromUpkeep`, calls originating from an authorized caller return `true`. This means that any strategist can exempt accounts they control from upkeep fees, circumventing the fee model.

Recommendation: Clarify the use of authorized caller array, if intended to permit who may call, a revision is needed. If intended to exempt Superform accounts from fees, access controls need to be adjusted to disallow Strategists from adding/removing accounts from the array.

Note that a strategist may act maliciously and add caller who expects to be receiving fees to the authorized caller array. They may even frontrun a transaction to do so.

The project team suggested noting accounts that should not be exempt as a potential solution. The following would be one way of doing so:

```
  function _isExemptFromUpkeep(
      address strategy,
      address updateAuthority,
      uint256 timestamp
  )
      internal
      returns (bool)
  {
      // ...snip...


      // Update is exempt if it is stale
      if (block.timestamp - timestamp > _strategyData[strategy].maxStaleness) {
          emit StaleUpdate(strategy, updateAuthority, timestamp);
          return true;
      }

      // ...snip...

+      // where `upkeepCallers` is a boolean mapping
+      if (upkeepCallers[updateAuthority]) { return true}

      // Check if the updateAuthority is in the authorized callers list
      uint256 authCallerLength = _strategyData[strategy].authorizedCallers.length;
      for (uint256 i; i < authCallerLength; i++) {
          if (_strategyData[strategy].authorizedCallers[i] == updateAuthority) {
              return true;
          }
      }

      return false;
  }
```

5.6.5 Consider a deadline that must pass before the owner may call `UpDistributor.reclaimTokens`

Severity: Informational

Context: UpDistributor.sol#L98

Description/Recommendation: Disallowing reclaiming of tokens before a deadline passes would give Up token recipients assurances that their claims will not be prematurely clawed back.

5.6.6 Avoid assigning a secondary strategist role when calling `changePrimaryStrategist`

Severity: Informational

Context: SuperVaultAggregator.sol#L411

Description/Recommendation: `_SUPER_GOVERNOR_ROLE` may replace the main strategist in case of emergency. Given an emergency is likely related to misuse, or loss of access to the account, adding the account automatically as a secondary strategist is not recommended.

Instead, simply replace the main strategist and if adding them back in a secondary role is desired, the new main strategist may do so in a separate transaction.

5.6.7 Consider greater separation being main strategist and secondary strategist capabilities

Severity: Informational

Context: SuperVaultAggregator.sol#L380

Description: The main strategist may update merkle roots to enable new hooks and restrict the secondary strategists to make risky calls. We recommend, similarly, disallowing secondary strategists from updating PPS Verification Thresholds. If no change to access is made, consider `isAnyStrategist` for consistency with other functions.

5.6.8 Consider ordered nonces

Severity: Informational

Context: SuperVault.sol#L232

Description/Recommendation: `SuperVault.authorizeOperator` uses unordered nonces. While the `deadline` offers some protection, setting and resetting of an authorization can be executed out of order in the case of a reorg or two transactions bundled after reading from a public mempool. An out of order pair of setting / resetting transactions would leave the contract with an authorization where none is intended.

5.6.9 Consider `DEADLINE_PASSED` instead of `TIMELOCK_NOT_EXPIRED`

Severity: Informational

Context: SuperVault.sol#L229

Description/Recommendation: `DEADLINE_PASSED`, or related error, more accurately describes what triggers this revert.

5.6.10 Consider adding a manual pause function for strategies

Severity: Informational

Context: SuperVaultAggregator.sol#L885-L894

Description/Recommendation: A strategy is only pausable as a byproduct of the outcome of PPS update checks, which means strategies cannot be manually paused in an emergency (and would be automatically unpaused as soon as a valid price update occurs). It is worth considering a manual pause function, accessible by the primary strategist, which would not be automatically reverted in `_forwardPPS`. This could be useful if, for example, the primary strategist believes the PPS validators are colluding against the strategy.

This would require either an additional pause property on the strategy (e.g. `isPaused` and `ppsUpdatesPaused`) or a reworking of the logic in `_forwardPPS` to skip paused strategies entirely.

5.6.11 Incorrect effective time check in `SuperGovernor`

Severity: Informational

Context: SuperGovernor.sol#L773-L776, SuperGovernor.sol#L811-L814

Description: In `SuperGovernor`, `executeAddIncentiveTokens()` and `executeRemoveIncentiveTokens()` check that the effective time for a proposal has passed as such:

```
if (
    _proposedAddWhitelistedIncentiveTokensEffectiveTime != 0
        && block.timestamp < _proposedAddWhitelistedIncentiveTokensEffectiveTime
) revert TIMELOCK_NOT_EXPIRED();
```

```
if (
    _proposedRemoveWhitelistedIncentiveTokensEffectiveTime != 0
        && block.timestamp < _proposedRemoveWhitelistedIncentiveTokensEffectiveTime
) revert TIMELOCK_NOT_EXPIRED();
```

However, these checks are incorrect as they allow both functions to be called when there is no proposal.

Recommendation: Both checks should be:

```
if (
    _proposedAddWhitelistedIncentiveTokensEffectiveTime == 0
        || block.timestamp < _proposedAddWhitelistedIncentiveTokensEffectiveTime
) revert TIMELOCK_NOT_EXPIRED();
```

5.6.12 `BaseHook` doesn't implement the `inspect()` function

Severity: Informational

Context: SuperVaultStrategy.sol#L1031-L1033

Description: When validating hooks in `SuperVaultStrategy._validateHook()`, it calls the hook's `inspect()` function:

```
return _getSuperVaultAggregator().validateHook(
    address(this), ISuperHookInspector(hook).inspect(hookCalldata), globalProof, strategyProof
);
```

This means all hooks to be used in strategies must implement an `inspect()` function. However, `BaseHook` does not implement the `inspect()` function, so new hooks created in the future may accidentally miss this function.

Recommendation: Refactor `BaseHook` to include the `inspect()` function.

5.6.13 Minor code improvements

Severity: Informational

Context: *(See each case below)*

Description/Recommendation:

1. ECDSAPPSOracle.sol#L148-L149: `validSignatureCount` will always be equal to `proofs.length` after the for-loop, there's no need for a separate variable. In the for-loop, `validSignatureCount` is equal to `i`.
2. SuperVault.sol#L28: Since `SuperVault` is a proxy, it should inherit the upgradeable versions of the OpenZeppelin contracts (i.e. `ERC20Upgradeable` and `ReentrancyGuardUpgradeable`). Additionally, consider inheriting `EIP712Upgradeable` for EIP-712 related logic, instead of writing your own implementation.

3. SuperGovernor.sol#L659-L668, SuperGovernor.sol#L697-L706: `proposeVaultBankHookMerkleRoot()` and `proposeSuperBankHookMerkleRoot()` are missing a `proposedRoot != bytes32(0)` check.
4. SuperGovernor.sol#L626-L634: In `executeUpkeepPaymentsChange()`, consider resetting `_proposedUpkeepPaymentsEnabled` to `false` for consistency.
5. SuperGovernor.sol#L108: `_SUPER_ASSET_FACTORY` is a contract key and not a role.
6. SuperGovernor.sol#L131-L134: The constructor is missing a zero address check for `prover_`.