SUPERFORM BETA

SuperForm Findings report

Date: 2023-11-03

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts. In a C4 solo audit a Warden, reviews, audits, or analyzes smart contract logic.
During the review outlined in this document, the C4 warden: Gerard Persoon conducted an analysis of the Superforms smart contract system written in Solidity. The review took place between September 18 and November 3, 2023, which included a fix review period.

Summary

The analysis yielded a total of 130 isssues. They can be subdived as follows:

- 3 vulnerabilities with a risk rating in the category of HIGH severity;
- 28 vulnerabilities with a risk rating in the category of MEDIUM severity;
- 21 vulnerabilities with a risk rating in the category of LOW severity.

Additionally, 52 informational issues have been found, as well as 26 gas improvement suggestions.
All issues have been address by the SuperForm project team. 13 issues are acknowledged and 117 issues are solved.

Scope

The reviewed code was stored in the following repositories:

- https://github.com/superform-xyz/superform-core/tree/2fa594b01e6c970200672a9b79018c11084032e6/src
- https://github.com/superform-xyz/ERC1155A/tree/2974a1c2d27885daaae51f822352733b38f61bd1/src

Disclaimer

Although the utmost effort has been put in this review by the Warden, it is no guarantee that all issues have been found and corrected.

Summary of issues

Severity: High Risk

H-1 : Funds might be recovered twice
H-2 : `||` is used instead of `&&`
H-3 : Function `ERC4626FormImplementation()` retrieved amount of tokens is different from swap input

Severity: Medium Risk

M-1 : Form Withdraw functions check swap token even where there is no swap
M-2 : Different IDs for Timelock forms
M-3 : `_burn()` doesn't work for the `isApprovedForAll` case
M-4 : Set functions don't remove old cross reference values
M-5 : Admin can disable himself via `setAddress()`
M-6 : Lowlevel call to EOA succeeds
M-7 : modifier `onlyForm()` can be circumvented
M-8 : Function `_processMultiDeposit()` could run out of gas
M-9 : Funds could get lost if `dstRefundAddress == 0`
M-10 : `proposeRescueFailedDeposits()` looses information of original values
M-11 : Return message might be send via fewer bridges
M-12 : `receiveMessage()` doesn't verify sender
M-13 : Insufficiently received proofs could lead to loss of funds
M-14 : `dispatchPayload()` could send too few proofs
M-15 : Broadcast message might be received on the same chain
M-16 : `_deployTransmuter()` allows duplicate deploy of `sERC20`
M-17 : No check `sERC20` has already been deployed
M-18 : No recovery mechanism for return messages
M-19 : `ERC4626KYCDaoForm` xchain functions check KYC token on xchain
M-20 : Function `_processDirectWithdraw()` could leave dust
M-21 : Limited slippage checks
M-22 : `_processDirectDeposit()` could leave dust
M-23 : `Forms` don't verify `vault` values
M-24 : No checks that `formBeaconId` corresponds to the `form`
M-25 : Forms are upgradable
M-26 : Forms are not authenticated
M-27 : Local deposits and withdraws don't check the paused state of forms
M-28 : Function `_singleXChainSingleVaultDeposit` sets an allowance twice

Severity: Low Risk

L-1 : Oracles could be stale
L-2 : Uptime of L2 influences oracles
L-3 : Different proof sizes in `PaymentHelper`
L-4 : Irrelevant `sERC20`s can be deployed
L-5 : Constant instead of function selector
L-6 : Delay might not be set yet
L-7 : Function `revokeRoleSuperBroadcast()` doesn't have a `nonce`
L-8 : No check of `dstPayloadId_` / `payloadId_`
L-9 : Non-initialization in for loop
L-10 : modifier `isValidPayloadId` missing
L-11 : Mint and burn asymmetric in `SuperTransmuter` / `Transmuter`
L-12 : Premissionless function `registerTransmuter()` can be abused
L-13 : `txHistory[]` not checked for validity
L-14 : Use of `transferFrom()` and `transfer()`
L-15 : Tighter checks on `_processDirectDeposit()` and `_processDirectWithdraw()` `liqDstChainId()`
L-16 : Types of input and output tokens not checked in `ERC4626FormImplementation()`
L-17 : `ERC4626FormImplementation()` retrieves `v.asset()` twice
L-18 : Constructor of `ERC4626FormImplementation()` doesn't check `stateRegistryId_`
L-19 : `block.chainid` might not fit in an uint64
L-20 : ChainId isn't checked for `0`
L-21 : PERMIT2 might not be set

Severity: Informational

I-1 : `estimateFees()` ignore unsupported chains
I-2 : Functions `_generateSingleVaultMessage()` and `_generateMultiVaultMessage()` use hardcoded value
I-3 : Address `DST_SWAPPER` retrieved inside a loop
I-4 : `AmbIds` in `_generateExtraData` hardcoded
I-5 : Is gas per byte or per kilobyte?
I-6 : Public function names start with _
I-7 : `TIMELOCK_FORM_ID` not used for xchain estimate functions
I-8 : functions `estimateMultiDstMultiVault()` only adds amount in the deposit case
I-9 : Prevent mistakes with `totalDstGas`
I-10 : Both functions `addChain()` and `updateChainConfig()` can do the same
I-11 : Code duplication between `Transmuter` and `ERC1155TokenReceiver`
I-12 : `synthethicTokenId[]== 0` not checked
I-13 : Use of `bridge` versus `ambId` is confusing
I-14 : Similar functions `hasProtocolAdminRole()` and `hasEmergencyAdminRole()` have different checks
I-15 : `revokeRoleSuperBroadcast()` and `stateSyncBroadcast()` derive address to revoke in different way
I-16 : Role configuration is very important
I-17 : Function `validateDepositPayloadUpdate()` and `validateWithdrawPayloadUpdate()` are similiar
I-18 : Function `packTxInfo()` can be changed to Solidity
I-19 : Combine two almost identical calls in `dispatchTokens()`
I-20 : `internal` function `dispatchTokens()` name doesn't start with an '_'.
I-21 : `LiquidityHandler` has two functions
I-22 : Function `processTx()` can call `dispatchTokens()`
I-23 : Incorrect/unexpected messages are return empty values
I-24 : No check of `timelockPayloadId_`
I-25 : `decodeStateSyncerPayloadHistory()` could revert on invalid data
I-26 : Function `decodeCoreStateRegistryPayload()` doesn't return all available data
I-27 : Validity check of `timeLockPayloadId_` in function `finalizePayload()`
I-28 : Terms `two step` and `timelock` both used
I-29 : Code duplication in `dispatchPayload()` and `_dispatchAcknowledgement()`
I-30 : Check in `_updateWithdrawPayload()` not obvious
I-31 : Indents can be reduced
I-32 : Incorrect comment in `_updateSingleVaultDepositPayload`
I-33 : Comments in `processPayload()` not accurate
I-34 : `receiveMessage()` uses a hardcoded value
I-35 : Functions `dispatchPayload()` and `broadcastPayload()` use a different pattern
I-36 : `_dispatchPayload()` and `_dispatchProof()` contain duplicate code
I-37 : `msg.sender` check could be modifier
I-38 : `stateMultiSync()` and `stateSync()` in `SuperPositions` and `SuperTransmuter` very similar
I-39 : No emit in `registerTransmuter()` and `_deployTransmuter()`
I-40 : Broadcast messages to new chains
I-41 : Comments about `multi` confusing
I-42 : `_kycCheck()` could be modifier
I-43 : Parameters for constructor of `ERC4626FormImplementation` not descriptive
I-44 : Reliance on `UPDATER_KEEPER`
I-45 : Incorrect/unexpected messages are ignored
I-46 : Shadowing variables `v.len` and `len`
I-47 : Inaccurate parameter name in function `_singleVaultTokenForward()`
I-48 : Retrieved addresses not checked for 0
I-49 : Paused values allow for mistakes
I-50 : Typos

I-51 : Functions `_validateSuperformData()` , `_validateSuperformsDepositData()` and `_validateSuperformsWithdrawData()` can be optimized
I-52 : Inconsistent placement of `burn` functions

Severity: Gas Optimization

G-1 : Functions `_getGasPrice()` and `_getNativeTokenPrice()` can be optimized
G-2 : Not all return data from `_estimateAMBFees()` is used
G-3 : Realistic input `estimateFees()` might not be neccesary
G-4 : Function `_generateExtraData()` can be optimized
G-5 : Unnecessary assignment in `estimateSingleDirectSingleVault()` and `estimateSingleDirectMultiVault()`,
G-6 : Function `estimateSingleDirectMultiVault()` can be optimized
G-7 : Function `estimateMultiDstMultiVault()` could use `len`
G-8 : An `emit` in function `setPermit2()` can be optimized
G-9 : `revokeRoleSuperBroadcast()` and `stateSyncBroadcast()` use different revoke functions
G-10 : Function `validateSuperformChainId()` can call `getDestinationChain()`
G-11 : Not all results of `getSuperforms()` are used
G-12 : `nonReentrant` is set and reset in a loop
G-13 : `Quorum` check done at end of function
G-14 : Function `finalizePayload()` sets status on memory copy
G-15 : Flags in `_processMultiDeposit()` can be set more efficient
G-16 : `_processMultiDeposit` can be optimized
G-17 : Use unchecked in function `_updateMultiVaultDepositPayload`
G-18 : For loops don't always cache length
G-19 : Function `processPayload()` can be optimized
G-20 : `synthethicTokenId[superformId_]` is first stored and then read again
G-21 : Nested `if` instead of `&&`
G-22 : Constants `PRECISION_DECIMALS` and `PRECISION` not used
G-23 : Assignment of `v.permit2` can be done only when necessary
G-24 : Function `_multiVaultTokenForward()` evaluates `vaultData_.liqData[0].token` twice
G-25 : Inconsistent checks for `amount_ == 0`
G-26 : Field `permit2` of struct `ValidateAndDispatchTokensArgs` is never used

Details of issues

Severity: High Risk

#H-1 : Funds might be recovered twice

Context: CoreStateRegistry.sol#L430-L548, CoreStateRegistry.sol#L713-L800
Description: Function `_updateMultiVaultDepositPayload()` potentially add an entry to `failedDeposits_[]`. After this function, the function `_processMultiDeposit()` is called which can also add an entry to `failedDeposits_[]`, for the same `i`.
A duplicate entry in `failedDeposits_[]` could mean the recover process will fail, due to checks on the array length. It can also mean the funds are payed back twice, effectly using funds from other users.
Note: this problem doesn't occur in `_processSingleDeposit(...) because` *updateSingleVaultDepositPayload()* `sets` *finalState* = PayloadState.PROCESSED`.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
   function _updateMultiVaultDepositPayload(...) ... {
       ...
       if (...) {
           ...
```

```
        } else {
                multiVaultData.amounts[i] = 0;
                failedDeposits_.superformIds.push(multiVaultData.superformIds[i]);
        }
        ...
    }
    function _processMultiDeposit(...) ... {
        ...
        try IBaseForm(superforms[i]).xChainDepositIntoVault(...) ... {
           ...
        ) returns (uint256 dstAmount) {
             ...
        } catch {
           ...
           failedDeposits[payloadId_].superformIds.push(multiVaultData.superformIds[i]);
        }
        ...
    }
}
```

Recommendation: In function `_processMultiDeposit(...)` skip processing it there is already a `failedDeposits[]` record. This can be done by checking `multiVaultData.amounts[i]==0`.
Superform: Solved in PR 254.
Reviewer: Verified

#H-2 : || is used instead of &&

Context: CoreStateRegistry.sol#L312-L336, PayloadUpdaterLib.sol#L67-L111, SuperRBAC.sol#L78-L98
Description: On several locations in the source `||` is used instead of `&&`. This is evaluated as one of the following:

- `(!a || !b) ==> !(a && b) ==> !(false) ==> true.`
- `(x!=a || x!=b) ==> !(x==a && x==b) ==> !(false) ==> true.`

This means the `if` will always be `true` and the code after it (usually `revert`) will always execute. Which means the function doesn't work as expected.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function disputeRescueFailedDeposits(uint256 payloadId_) external override {
        ...
        /// @dev the msg sender should be the refund address (or) the disputer
        if (
            msg.sender != failedDeposits_.refundAddress
                || !_hasRole(keccak256("CORE_STATE_REGISTRY_DISPUTER_ROLE"), msg.sender)
        ) {
            revert Error.INVALID_DISUPTER();   // always reverts
        }
        ...
    }
}
```

```
library PayloadUpdaterLib {
    function validateDepositPayloadUpdate(...) ... {
        ...
        if (txType != uint256(TransactionType.DEPOSIT) || callbackType != uint256(CallbackType.INIT)
            revert ...;
        }
    }
    function validateWithdrawPayloadUpdate(...) ... {
        ...
        if (txType != uint256(TransactionType.WITHDRAW) || callbackType != uint256(CallbackType.INIT
            revert ...;
        }
    }
}
```

```
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    function revokeRoleSuperBroadcast(...) ... {
        ...
        if (role_ != PROTOCOL_ADMIN_ROLE || role_ != EMERGENCY_ADMIN_ROLE)
            revokeRole(role_, addressToRevoke_);
        ...
    }
```

Recommendation: Doublecheck all `||` and `&&` expressions and fix them.
Superform: Solved in PR 233 and PR 242.
Reviewer: Verified

#H-3 : Function `ERC4626FormImplementation()` retrieved amount of tokens is different from swap input

Context: ERC4626FormImplementation.sol#L108-L173, LiquidityHandler.sol#L24-L53
Description: When function `ERC4626FormImplementation()` wants to swap tokens, it retrieves `singleVaultData_.amount` tokens. But then it swaps this amount: `IBridgeValidator(vars.bridgeValidator).decodeAmountIn(singleVaultData_.liqData.txData, false)`. These values are most likely different, which leads to reverts later on in the code.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (ui
        ...
        token.safeTransferFrom(msg.sender, address(this), singleVaultData_.amount);
        if (singleVaultData_.liqData.txData.length > 0) {
             ...
             dispatchTokens(..., IBridgeValidator(vars.bridgeValidator).decodeAmountIn(singleVaultDa
        }
        ...
    }
```

```
abstract contract LiquidityHandler {
    function dispatchTokens(..., uint256 amount_,...) ... {
        ...
        token.safeIncreaseAllowance(bridge_, amount_);
        ... // call bridge
    }
```

Recommendation: In function `_processDirectDeposit()`, when swapping, retrieve the same amount of tokens that are swapped.
Superform: Solved in PR 215, commit 3312ed2.
Reviewer: Verified

Severity: Medium Risk

#M-1 : Form Withdraw functions check swap token even where there is no swap

Context: ERC4626FormImplementation.sol#L185-L240, ERC4626FormImplementation.sol#L276-L339
Description: As found by the Superform project: The functions `_processDirectWithdraw()` and `_processXChainWithdraw()` use `singleVaultData_.liqData.token` to compare to the `collateral`, in all cases. This should only happen when a swap is done (e.g. when `len != 0`). This way all withdraws without a swap will fail.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function _processDirectWithdraw(...) ... {
        v.len1 = singleVaultData_.liqData.txData.length;

        /// @dev the token we are swapping from to our desired output token (if there is txData), mu
        /// the vault asset
        if (singleVaultData_.liqData.token != v.collateral) revert Error.DIRECT_WITHDRAW_INVALID_COL
        ...
        if (v.len1 != 0) {
```

```
            ... // swap singleVaultData_.liqData.token to ...
        }
    }
    function _processXChainWithdraw(...) ... {
        uint256 len = singleVaultData_.liqData.txData.length;

        /// @dev the token we are swapping from to our desired output token (if there is txData), mu
        /// the vault asset
        if (vars.collateral != singleVaultData_.liqData.token) revert Error.XCHAIN_WITHDRAW_INVALID_
        ...
        if (len != 0) {
            ... // swap singleVaultData_.liqData.token
        }
    }
}
```

Recommendation: Move the check between `singleVaultData_.liqData.token` and `collateral`, inside the `if (len !=0)` statement.
Superform: Solved in PR 246 and PR 319.
Reviewer: Verified

#M-2 : Different IDs for Timelock forms

Context: PaymentHelper.sol#L33-L34, Abstract.Deploy.s.sol#L148-L150
Description: The contract `PaymentHelper` defines sets `TIMELOCK_FORM_ID = 1` , however in the testfiles the form id for `ERC4626TimelockForm ==2`. This could mean the tests are not accurate and possibly the deployment isn't accurate.

```
contract PaymentHelper is IPaymentHelper {
    uint32 public constant TIMELOCK_FORM_ID = 1;
    ...
}
```

```
abstract contract AbstractDeploy is Script {
    /// @dev 1 = ERC4626Form, 2 = ERC4626TimelockForm, 3 = KYCDaoForm
    uint32[] public FORM_IMPLEMENTATION_IDS = [uint32(1), uint32(2), uint32(3)];
    string[] public VAULT_KINDS = ["Vault", "TimelockedVault", "KYCDaoVault"];
}
```

Recommendation: Doublecheck the form Id for timelock forms.
Superform: Solved in PR 280.
Reviewer: Verified

#M-3 : `_burn()` doesn't work for the `isApprovedForAll` case

Context: ERC1155A.sol#L438-L486
Description: As found by the Superform project: Function `_burn()` always lowers `allowances`. In function `_batchBurn()` , when `isApprovedForAll[]==true` then `singleApproval==false` then no `allowances` are lowered. This means `_burn()` doesn't work for the `isApprovedForAll` case.

```
abstract contract ERC1155A is IERC1155A {
    function _batchBurn(address from, uint256[] memory ids, uint256[] memory amounts) internal virtu
        ...
        if (msg.sender != from && !isApprovedForAll[from][msg.sender]) {
            singleApproval = true;
        }
        ...
        if (singleApproval) {
            require(allowance(from, msg.sender, id) >= amount, "NOT_AUTHORIZED");
            allowances[from][msg.sender][id] -= amount;
        }
        ...
    }
    function _burn(address from, uint256 id, uint256 amount) internal virtual {
```

```
        if (msg.sender != from && !isApprovedForAll[from][msg.sender]) {
            require(allowance(from, msg.sender, id) >= amount, "NOT_AUTHORIZED");
        }
        allowances[from][msg.sender][id] -= amount;
        ...
    }
}
```

Recommendation: Change the code of `_burn()` to something like this:

```
function _burn(address from, uint256 id, uint256 amount) internal virtual {
    if (msg.sender != from && !isApprovedForAll[from][msg.sender]) {
        require(allowance(from, msg.sender, id) >= amount, "NOT_AUTHORIZED");
+       allowances[from][msg.sender][id] -= amount;
    }
-   allowances[from][msg.sender][id] -= amount;
    ...
}
```

Superform: Solved by ERC1155A PR 20.
Reviewer: Verified

#M-4 : Set functions don't remove old cross reference values

Context: SuperRegistry.sol#L143-L225, LayerzeroImplementation.sol#L92-L112
Description: Several functions update values with crossreferences, which could lead to issues if old values are not cleaned up. Here is an example:

```
forward[1] = 0x1234;
backward[0x1234] = 1;
```

Then when an update (correction) is made:

```
forward[1] = 0x5678;
backward[0x5678] = 1;
```

Now if you look up `backward[0x1234]` you get `1` and if you look up `forward[1]` you get `0x5678`, which is not consistent.
This problem occurs with the following functions:

```
contract SuperRegistry is ISuperRegistry, QuorumManager {

   function setAmbAddress(...) ... {
       ...
       ambAddresses[ambId] = ambAddress;
       ambIds[ambAddress] = ambId;
       isBroadcastAMB[ambId] = broadcastAMB;
       ...
   }
   function setStateRegistryAddress(...) ... {
       ...
       registryAddresses[registryId] = registryAddress;
       stateRegistryIds[registryAddress] = registryId;
       ...
   }
   function setRouterInfo(...) ... {
       ...
       stateSyncers[superFormRouterId] = stateSyncer;
       routers[superFormRouterId] = router;
       superformRouterIds[router] = superFormRouterId;
       ...
   }
}
```

```
contract LayerzeroImplementation is IAmbImplementation, ILayerZeroUserApplicationConfig, ILayerZeroR
    function setChainId(uint64 superChainId_, uint16 ambChainId_) external onlyProtocolAdmin {
        ...
```

```
        /// @dev  reset old mappings
        uint64 oldSuperChainId = superChainId[ambChainId_];
        uint16 oldAmbChainId = ambChainId[superChainId_];
        if (oldSuperChainId > 0) {
            ambChainId[oldSuperChainId] = 0;
        }
        if (oldAmbChainId > 0) {
            superChainId[oldAmbChainId] = 0;
        }
        ambChainId[superChainId_] = ambChainId_;
        superChainId[ambChainId_] = superChainId_;
    }
}
```

Recommendation: Use the following approach to first remove old values: Note: this pattern is also used in `setChainId()` of `LayerzeroImplementation`. Alternatively, for values that only need to be set once, check there are no previous values.

```
oldForward = forward[...];
oldBackward = backward[...];
if (oldForward !=0 ) {
    delete backward[oldForward];
}
if (oldBackward !=0 ) {
    delete forward[oldBackward];
}
```

Superform: Solved in PR 233 and PR 259.
Reviewer: Verified

#M-5 : Admin can disable himself via `setAddress()`

Context: SuperRegistry.sol#L67-L72, SuperRegistry.sol#L107-L111
Description: Function `setAddress()` can update the `registry[SUPER_RBAC][uint64(block.chainid)]` value. This value is used in `modifier onlyProtocolAdmin()`. Such an (accidental) update could disable access for the ProtocolAdmin and thus disable future updates (e.g. shoot yourself in the foot). Depending on the new `SUPER_RBAC`, also more ProtocolAdmins could be added this way.
For comparison: contract `SuperRBAC` takes precautions to prevent deleting the last admin.

```
contract SuperRegistry is ISuperRegistry, QuorumManager {
    modifier onlyProtocolAdmin() {
        if (!ISuperRBAC(registry[SUPER_RBAC][uint64(block.chainid)]).hasProtocolAdminRole(msg.sender
            revert Error.NOT_PROTOCOL_ADMIN();
        }
        _;
    }
    function setAddress(bytes32 id_, address newAddress_, uint64 chainId_) external override onlyPrc
        ...
        registry[id_][chainId_] = newAddress_;
        ...
    }
}
```

Recommendation: Add extra checks, for example verify that the admin still has access after changing the SUPER_RBAC address.
Superform: Solved in PR 233 by only allowing to set SUPER_RBAC once.
Reviewer: Verified

#M-6 : Lowlevel call to EOA succeeds

Context: LiquidityHandler.sol#L24-L53
Description: If `bridge_ == address(0)` or any other EOA then the lowlevel call in function `dispatchTokens()` will succeed. Usually `dispatchTokens()` is called with the results of

`superRegistry.getBridgeAddress(...)`, which could return `address(0)`. Sending tokens to `address(0)` or an EOA which isn't accessible, will result in the loss of tokens.

```
abstract contract LiquidityHandler {
     function dispatchTokens(...) ... {
         ...
        (bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
        ...
    }
}
```

Here is a POC showing this behaviour:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.21;
import "hardhat/console.sol";

contract test {
    constructor() {
        bytes memory txData_;
        (bool success,) = payable(address(0)).call{ value: 0 }(txData_);
        console.log("success",success);
    }
}
```

Recommendation: Check `bridge_` isn't `address(0)` and/or check `bridge_` is a contract. Also see this issue about checking that the result of `getBridgeAddress()` isn't `address(0)`: Retrieved addresses not checked for 0.
Superform: Solved in PR 249.
Reviewer: Verified

#M-7 : modifier `onlyForm()` can be circumvented

Context: TimelockStateRegistry.sol#L65-L72, TimelockStateRegistry.sol#L91-L106, DataLib.sol#L48-L56
Description: Function `receivePayload()` has a modifier `onlyForm()`. However the check `msg.sender != superform` isn't effective, because the `superformId` is supplied to `receivePayload()` via the parameter `data_`. Any caller to `receivePayload()` can fill in anything there. The next check with `getStateRegistryId()` can also be circumvented because a user supplied `superform` contract could return any value for `getStateRegistryId()`.
The followup actions (redeeming and minting superPositions) are linked to the malicious `form`. However a `_dispatchAcknowledgement()` could be send with any `payloadId`. This could result in a denial of service / funds loss if the suggestion of this issue is implemented: `txHistory[]` can be reset after use.

```
contract TimelockStateRegistry is BaseStateRegistry, ITimelockStateRegistry, ReentrancyGuard {
     modifier onlyForm(uint256 superformId) {
        (address superform,,) = superformId.getSuperform();
        if (msg.sender != superform) revert Error.NOT_SUPERFORM();
        if (IBaseForm(superform).getStateRegistryId() != superRegistry.getStateRegistryId(address(th
            revert Error.NOT_TWO_STEP_SUPERFORM();
        }
        _;
    }
    function receivePayload(..., InitSingleVaultData memory data_) ... {
        ++timelockPayloadCounter;
        timelockPayload[timelockPayloadCounter] =
            TimelockPayload(type_, srcSender_, srcChainId_, lockedTill_, data_, TwoStepsStatus.PENDI
    }
}
library DataLib {
    function getSuperform(uint256 superformId_) ... {
        superform_ = address(uint160(superformId_));
        formBeaconId_ = uint32(superformId_ >> 160);
        chainId_ = uint64(superformId_ >> 192);
    }
```

Recommendation: Consider checking the superform address exists on superform factory.
Superform: Solved by PR 297.
Reviewer: Verified

#M-8 : Function `_processMultiDeposit()` could run out of gas

Context: CoreStateRegistry.sol#L713-L800
Description: Function `_processMultiDeposit()` could run out of gas if too many deposits are done. Depending on the moment it occurs the entire call `_processMultiDeposit` reverts and also the updates to `failedDeposits[]` revert. This means the recovery mechanism doesn't work.
Potential workaround: set all the `forms` on `paused` then the calls to `xChainDepositIntoVault` will quickly revert, however this is unwanted because that would be a denial of service for other users of the protocol.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _processMultiDeposit(...) ... {
        ...
        for (uint256 i; i < numberOfVaults;) {
            ...
            try IBaseForm(superforms[i]).xChainDepositIntoVault(...) returns (uint256 dstAmount) {
                ...
            } catch {
               ...
               failedDeposits[payloadId_].superformIds.push(multiVaultData.superformIds[i]);
            }
       }
    }
}
```

Recommendation: Have a way to partially process the deposits and/or have a way to recover funds even if `_processMultiDeposit()` always fails.
Superform: Solved in PR 308.
Reviewer: Verified

#M-9 : Funds could get lost if `dstRefundAddress == 0`

Context: CoreStateRegistry.sol#L274-L309
Description: Function `proposeRescueFailedDeposits()` assigns a `refundAddress`, but doesn't check for `0`. If `dstRefundAddress == 0` then funds send to this address would get lost.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function proposeRescueFailedDeposits(...) ... {
        ...
         failedDeposits_.refundAddress = data.dstRefundAddress;
        ...
    }
}
```

Recommendation: Do one of the following:

- check that and enforce `dstRefundAddress` is present when submitting a transaction
- use `srcSender` if the `dstRefundAddress` is `address(0)`. Note: this could be an issue with smart contract wallets, which might not be deployed on the xchain.

Superform: Solved by PR 257.
Reviewer: Verified

#M-10 : `proposeRescueFailedDeposits()` looses information of original values

Context: CoreStateRegistry.sol#L274-L309
Description: Function `proposeRescueFailedDeposits()` overwrites `failedDeposits_.amounts` and there is no relation of the to-be-rescued-amounts to the original amounts. So the function could rescue more than is lost,

which would use funds from other users. If not enough funds are rescued, then the funds stay in the `CoreStateRegistry` contract. The updates also complicate invariant checks.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function proposeRescueFailedDeposits(...) ... {
        ...
        failedDeposits_.amounts = proposedAmounts_;
        ...
    }
}
```

Recommendation: Consider keeping the original amounts `_updateMultiVaultDepositPayload()` and compare the amounts.
Superform: On this issue, the amount checks could lead to a false sense of security (or) at some cases can push the protocol into irrecoverable state.

- There are case where this might go wrong, if we check the rescuer input amount against the user input, then the check could be totally irrelevant. since the rescuer hacker can input any random values in superformdata.amount and try to attack the protocol.
- The other case is, if we check the rescuer input amounts against UPDATER amounts and make the UPDATER amounts as the higher threshold, then that could lead the protocol into irrecoverable state, since UPDATER can even set the amount to zero.

Right now, the process is RESCUER propose any arbitrary value and the DISPUTER checks it over the Timelock period to ensure it's a right value. Doing this check on chain could be dangerous provided the real amount is still known offchain.
While we agree with failedDeposits_.amounts being a helpful measure if the user is not malicious, this number is something that can be spoofed and as an upper bound is not particularly helpful to the protocol. The propose/dispute process additionally protects the user from a corrupted UPDATER/RESCUER. If the user receives more than the deposit.amount, only that number is deposited anyway as well. We think that loosing this information is fine for the protocol but acknowledge there could be other ways to run this process!
Reviewer: Acknowledged

#M-11 : Return message might be send via fewer bridges

Context: CoreStateRegistry.sol#L81-L135, CoreStateRegistry.sol#L188-L271
Description: Assume there are 3 bridges used for proofs and the quorum is set at 2. Then when 2 messages are received, `updateDepositPayload()` can run. There still could be a proof message on its way from the 3rd bridge. Once the 3rd proof is received it is stored in `proofAMB[prevPayloadProof]` and will not be added to `proofAMB[newPayloadProof]`. So `proofAMB[newPayloadProof]` stays at 2 and the return message is send back over fewer bridges then the original message.
Normally the 2 bridges for proof should be sufficient to the deliver the message, but it is less robust. Also see issue: No recovery mechanism for return messages.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function updateDepositPayload(...) ... {
        ...
        if (messageQuorum[prevPayloadProof] < _getRequiredMessagingQuorum(srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
        if (newPayloadProof != prevPayloadProof) {
            messageQuorum[newPayloadProof] = messageQuorum[prevPayloadProof];
            proofAMB[newPayloadProof] = proofAMB[prevPayloadProof];
            ...
        }
        ...
    }
    function processPayload(uint256 payloadId_) ... {
        ...
        uint8[] memory proofIds = proofAMB[v._proof];
```

```
        ...
        uint8[] memory ambIds = new uint8[](proofIds.length + 1);
        ambIds[0] = msgAMB[payloadId_];
        uint256 len = proofIds.length;
        for (uint256 i; i < len;) {
            ambIds[i + 1] = proofIds[i];
            ...
        }
        _dispatchAcknowledgement(v.srcChainId, ambIds, returnMessage);
       ...
    }
}
```

Recommendation: Include the received messages, even if they are received after `updateDepositPayload()`. This will require some changes in the datastructure.
Superform: Solved by sending along the list of ambs in PR 316.
Reviewer: Verified

#M-12 : `receiveMessage()` doesn't verify sender

Context: WormholeSRImplementation.sol#L105-L133
Description: Function `receiveMessage()` doesn't validate the sender on the source chain. However as relayers are moving towards permissionlessness a malicious or buggy relayer might call `receiveMessage()` with an incorrect message that doesn't originate from the protocol. This could lead to:

- incorrectly pausing formbeacon, leading to denial of service;
- incorrectly deploying eERC20s, possibly making funds inaccessible;
- incorrectly removing roles, leading to denial of service.

```
contract WormholeSRImplementation is IBroadcastAmbImplementation {
    function receiveMessage(bytes memory encodedMessage_) public {
         ...
        /// @dev 2. validate src chain sender
        ...
    }
}
```

Recommendation: Validate the source chain sender, possibly by verifying the caller from `IWormhole.VM emitter` address.
Superform: Solved in PR 304.
Reviewer: Verified

#M-13 : Insufficiently received proofs could lead to loss of funds

Context: CoreStateRegistry.sol#L81-L271
Description: All the functions that could potentially result in sending a message back or updating `failedDeposits[]` have a check to verify enough proofs are received. If insufficient proofs are received, because insufficient proofs are send, or because some proofs are not delivered, then no message is send back and `failedDeposits[]` is never updated. This means the transmitted funds cannot be rescued. Also see issue `dispatchPayload()` could send too few proofs.
Note: a workaround would be add a router via `setRouterInfo()` which could call `dispatchPayload()` with the same message to transfer it over additonal `ambIds`, however this is an unwanted solution. Another workaround would be to lower the `Quorum`. This might be undesirable too. Probably you would want to replace the bridge with a functioning bridge and keep the `Quorum` present.
Note: the same issue can occur on return messages, see issue No recovery mechanism for return messages.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
   function updateDepositPayload(
        ...
        if (messageQuorum[prevPayloadProof] < _getRequiredMessagingQuorum(srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
```

```
        }
        ...
    }
    function updateWithdrawPayload(...) ... {
        ...
        if (messageQuorum[v.prevPayloadProof] < _getRequiredMessagingQuorum(v.srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
    }
    function processPayload(...) ... {
        ...
        if (messageQuorum[v._proof] < _getRequiredMessagingQuorum(v.srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
    }
}
```

Recommendation: Have a way to rescue funds, even if insufficient proofs are received.

Superform: We recommend users to send as many proofs as we support for reliability (although they still have a single point of failure on the main AMB bridge). But if this case particularly were to happen, we could investigate if reducing the quorum (proof bridges) could allow for normal transactions to go through, albeit admittedly not the most secure method for users who weren't impacted by the initial quorum.

We are concerned that implementing a rescue mechanism would not have the intended impact because any rescue/return withdraw message would not be secure and also get sent via the same (possibly broken/corrupt) AMBs that the original message was sent on.

Reviewer: Acknowledged

#M-14 : `dispatchPayload()` could send too few proofs

Context: BaseStateRegistry.sol#L68-L87, CoreStateRegistry.sol#L188-L271, CoreStateRegistry.sol#L947-L958, TimelockStateRegistry.sol#L296-L306, BroadcastRegistry.sol#L98-L117

Description: The function `dispatchPayload()` sends proofs even if only one `ambId` is specified. On the `xchain`, function `processPayload()` only continues processing if at least `_getRequiredMessagingQuorum()` proofs have been received.

The same issue is present in `broadcastPayload()`. A similar issue is present with `_dispatchAcknowledgement()` of both `CoreStateRegistry` and `TimelockStateRegistry`, although that won't occur in practice, because the return `ambIds` are retrieved from the received `ambIds`, which are checked to be `>= _getRequiredMessagingQuorum(...)`.

Also see issue: Insufficiently received proofs could lead to loss of funds.

```
abstract contract BaseStateRegistry is IBaseStateRegistry {
    function dispatchPayload(...) ... {
        ...
        if (ambIds_.length > 1) {
            _dispatchProof(...);
        }
    }
}
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function processPayload(uint256 payloadId_) ... {
        ...
        if (messageQuorum[v._proof] < _getRequiredMessagingQuorum(v.srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
    }
    function _dispatchAcknowledgement(uint64 dstChainId_, uint8[] memory ambIds_, bytes memory messa
        ...
        if (ambIds_.length > 1) {
            _dispatchProof(...);
        }
    }
```

```
}
contract BroadcastRegistry is IBroadcastRegistry, QuorumManager {
    function broadcastPayload(...) ... {
        ...
        if (ambIds_.length > 1) {
             ...
            _broadcastProof(...);
        }
    }
}
```

Recommendation: Make sure at least `_getRequiredMessagingQuorum()` proofs are send to different `ambIds`.
Superform: Solved in PR 316.
Reviewer: Verified

#M-15 : Broadcast message might be received on the same chain

Context: WormholeSRImplementation.sol#L77-L133
Description: The broadcasting works in the following way: the relayers generate VAAs and superform relays the message. The relayers should only transmit to `other` chains and call `receiveMessage()` on these chains.
However as relayers are moving towards permissionlessness a malicious or buggy relayer could call `receiveMessage()` on the originating chain. There is no explicit check in `receiveMessage()` to prevent this. Then the following would happen:

- `changeFormBeaconPauseStatus()` would set the paused state twice (direct and via de broadcast); In combination with race conditions, setting to paused & unpaused quickly might result in unexpected final pause status;
- `registerTransmuter()` would create an `serc20 twice` (once direct and once via de broadcast), also see issue `_deployTransmuter()` allows duplicate deploy of `sERC20`.
- `revokeRoleSuperBroadcast` would do `revokeRole()` twice

```
contract WormholeSRImplementation is IBroadcastAmbImplementation {
    function broadcastPayload(...) ... {
       ...
        wormhole.publishMessage{ value: msg.value }(...);
    }
   function receiveMessage(bytes memory encodedMessage_) public {
        ...
        if (processedMessages[wormholeMessage.hash]) {
            revert Error.DUPLICATE_PAYLOAD();
        }
        processedMessages[wormholeMessage.hash] = true;
        ...
    }
}
```

Recommendation: Consider setting `processedMessages[wormholeMessage.hash] = true;` in `broadcastPayload()` if it is possible the determine the hash. This way `receiveMessage()` won't process the message if it is accidentally received on the same chain. Alternative check that the received messages didn't originate from the same chain.
Superform: Solved in PR 281.
Reviewer: Verified

#M-16 : `_deployTransmuter()` allows duplicate deploy of `sERC20`

Context: SuperTransmuter.sol#L92-L125, SuperTransmuter.sol#L329-L342
Description: The function `registerTransmuter()` checks a `sERC20` hasn't been deployed yet. However `_deployTransmuter()` doesn't explicitly check this. A duplicate call to `_deployTransmuter()` shouldn't happen, but if it does its probably better not to deploy a new token. If this would happen after some `sERC20` already

have been minted, then the `old sERC20` couldn't be burnt and minted. This will also complicate invariant checks. Also see issues:

- Broadcast message might be received on the same chain
- No recovery mechanism for return messages that can't be processed.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(uint256 superformId_, bytes memory extraData_) external override ret
        ...
        if (synthethicTokenId[superformId_] != address(0)) revert TRANSMUTER_ALREADY_REGISTERED();
        ...
        synthethicTokenId[superformId_] = address(new sERC20(...));
        ...
    }
    function _deployTransmuter(bytes memory message_) internal {
        ...
        address syntheticToken = address(new sERC20(...));
        synthethicTokenId[superformId] = syntheticToken;
        ...
    }
```

Recommendation: In function `_deployTransmuter()`, revert when `synthethicTokenId[superformId]!=0`.
Superform: Solved in PR 248.
Reviewer: Verified

#M-17 : No check `sERC20` has already been deployed

Context: SuperTransmuter.sol#L197-L295
Description: The functions `stateMultiSync()` and `stateSync()` of `SuperTransmuter` don't check the `sERC20` has already been deployed. The `sERC20` could be absent if:

- `registerTransmuter()` hasn't been called yet;
- `registerTransmuter()` hasn't properly transmitted the data to other chains, see issue Premissionless function `registerTransmuter()` can be abused;
- a new chain is added on which the `sERC20` token is not depoyed yet, see issue Broadcast messages to new chains.

It is also difficult to recover from this situation, see issue No recovery mechanism for return messages that can't be processed.
If the `sERC20` hasn't been deployed then the `mint()` will fail, however this will be more difficult to debug.

```
function stateMultiSync(AMBMessage memory data_)
    ...
    sERC20(synthethicTokenId[returnData.superformIds[i]]).mint(srcSender, returnData.amounts[i]);
    ...
}
function stateSync(AMBMessage memory data_)
    ...
    sERC20(synthethicTokenId[returnData.superformId]).mint(srcSender, returnData.amount);
    ...
}
```

Recommendation: When starting a deposit transaction consider checking a `sERC20` token has been deployed and revert if it hasn't. In functions `stateMultiSync()` and `stateSync()` consider checking a `sERC20` token has been deployed and give an approriate error message on the revert.
Superform: Solved in PR 247.
Reviewer: Verified

#M-18 : No recovery mechanism for return messages

Context: CoreStateRegistry.sol#L188-L271, TimelockStateRegistry.sol#L186-L218

Description: The function `processPayload()` of `CoreStateRegistry` and `TimelockStateRegistry` processes return messages. However if somehow insufficient proofs are received or the functions `stateMultiSync()` or `stateSync()` fail, then there is no recovery mechanism and the tokens might not get minted. This way funds are inaccessible. For comparison, if transactions fail on the xchain fail there is a recovery mechanism.
Also see issues:

- Insufficiently received proofs could lead to loss of funds
- No check `sERC20` has already been deployed.

A manual recovery mechanism exists by assigning the `onlyMinter` rights and manualling minted tokens. However this has its own risks see issue TBD.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function processPayload(uint256 payloadId_)
        ...
        if (messageQuorum[v._proof] < _getRequiredMessagingQuorum(v.srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
        /// @dev mint superPositions for successful deposits or remint for failed withdraws
        if (v.callbackType == uint256(CallbackType.RETURN) || v.callbackType == uint256(CallbackType
            v.multi == 1
                ? IStateSyncer(_getStateSyncer(abi.decode(v._payloadBody, (ReturnMultiData)).superfo
                    .stateMultiSync(_message)
                : IStateSyncer(_getStateSyncer(abi.decode(v._payloadBody, (ReturnSingleData)).superf
                    _message
                );
        }
        ...
    }
}
```

Recommendation: Design a recovery mechanism, comparable to the recovery mechanism in `CoreStateRegistry`.
Superform: We recommend users to send as many proofs as we support for reliability (although they still have a single point of failure on the main AMB bridge). But if this case particularly were to happen, we could investigate if reducing the quorum (proof bridges) could allow for normal transactions to go through, albeit admittedly not the most secure method for users who weren't impacted by the initial quorum.
We are concerned that implementing a rescue mechanism would not have the intended impact because any rescue/return withdraw message would not be secure and also get sent via the same (possibly broken/corrupt) AMBs that the original message was sent on. Plus we now mandate on src that the appropriate number of AMB's are sent via quorum directly on src, so insufficiently received proofs should be impossible unless the bridge is down.
And particular cases where `stateSync()` could be failing addressed here: PR 308
Reviewer: Acknowledged

#M-19 : `ERC4626KYCDaoForm` xchain functions check KYC token on xchain

Context: ERC4626KYCDaoForm.sol#L33-L35, ERC4626KYCDaoForm.sol#L65-L92, kycdao4626.sol#L175-L177
Description: When using `ERC4626KYCDaoForm` in combination with `xchain` deposits or withdraws, the KYC token is checked on the `xchain`, while `srcSender` is on the local chain. With smart contract wallets, it might be difficult to have the same address on the xchain. The mechanism could potentially be misused if an attacker manages to deploy a smart contract wallet on the source chain with the same address as a smart contract wallet on the xchain.

```
contract kycDAO4626 is ERC4626 {
    function kycCheck(address user_) public view returns (bool) {
        return kycValidity.hasValidToken(user_);
    }
```

```
contract ERC4626KYCDaoForm is ERC4626FormImplementation {
    function _kycCheck(address srcSender_) internal view {
        if (!kycDAO4626(vault).kycCheck(srcSender_)) revert NO_VALID_KYC_TOKEN();
    }
    function _xChainDepositIntoVault(...) ... {
        _kycCheck(srcSender_);
        ...
    }
    function _xChainWithdrawFromVault(...) ... {
        _kycCheck(srcSender_);
        ...
    }
```

Recommendation: Some suggestions to support smart contract wallets:

1. `isValidSignature()` of https://eips.ethereum.org/EIPS/eip-1271 might be used where the smart contract account on the xchain verifies a signature of a (hashed) SrcSender In practice that might be cumbersome the achieve: first you would have to sign something on the `xchain` and then add that info to call on the source chain. Then you might as well do the transaction directly on the `xchain`
2. if the smart contract wallet has an `owner()` function or something like that, that function could be called on the `xchain` and the result could be compared to the `SrcSender`. However there is no standard for this afaik and with a `multisig` it is probably not enough that just one of the owners is verified.
3. the sender of the transaction can supply a second address (e.g. the smart wallet address on the `xchain`), similar to `refundAddress`. Then the `superPostion` tokens are also send to the smart wallet on the `xchain`. For withdraw the smart wallet address on the `xchain` has to set an allowance to do it in the reverse order. Note: this way you could still abuse a smart contract wallet from someone else on the deposit.
4. let the smart contract wallet on the `xchain` call a function on the `ERC4626KYCDaoForm` where it explicitly links the `SrcSender` to a smart contract wallet - then that smart contract wallet can be used to check it has a KYCDao token.

Superform: We've decided to not launch with this `Form` and wait to see how exactly we want to support KYC yield on `Superform`. While `KYCDAO` was the implementation we decided to build around here, others use things like whitelists, and we want to make sure that whatever design we choose can be applied more broadly. It's possible that to do this securely deposits and withdrawals must be done same-chain to avoid the misrepresentation of KYC and we'd have to restrict the Form accordingly (assuming the KYC isn't natively on the other chain too). Some combination of idea 2 & 3 made possible with free read calls could work to support this later on.
Reviewer: Acknowledged

#M-20 : Function `_processDirectWithdraw()` could leave dust

Context: ERC4626FormImplementation.sol#L185-L240
Description: In function `_processDirectWithdraw()`, it could happen that the amount of redeemed tokens (`dstAmount`) is larger than the specified input amount for a swap (`v.amount`), so `v.amount < dstAmount`. Then some tokens are not swapped and not send to `srcSender`. They stay in the form contract as dust. In extreme situations (sudden changes in token price, perhaps caused by a sandwich), this could be a larger amount of tokens.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function _processDirectWithdraw(...) ... {
        ...
        dstAmount = v.v.redeem(singleVaultData_.amount, v.receiver, address(this));
        if (v.len1 != 0) {
            v.amount = IBridgeValidator(v.bridgeValidator).decodeAmountIn(singleVaultData_.liqData.t

            /// @dev the amount inscribed in liqData must be less or equal than the amount redeemed
            if (v.amount > dstAmount) revert Error.DIRECT_WITHDRAW_INVALID_LIQ_REQUEST();
            ....
        }
    }
}
```

Recommendation: Consider checking the amount of dust, possibly via a slippage parameter. Revert if this is too large. Alternatively leave dust if it is small and transfer it to the Paymaster if it is larger.
Superform: Solved in PR 252.
Reviewer: Verified

#M-21 : Limited slippage checks

Context: CoreStateRegistry.sol#L430-L711, DstSwapper.sol#L70-L134
Description: Slippage could potentially occur on swaps, bridging and depositing/redeeming from vaults. Note: attackers could perhaps manipulate the erc4626 vaults by sandwiching them. A way to counter that would be to do a slippage check.
There is some code on the destination chain which references `maxSlippage`. This is checked with `_updateMultiVaultDepositPayload()`, `_updateSingleVaultDepositPayload()` and `_processMultiWithdrawal()`. It is supplied to `xChainWithdrawFromVault()`, but this function doesn't reference `maxSlippage`.
The `DstSwapper` could result in slippage via the swaps that are started via `batchProcessTx()` / `processTx()`. However this is only detected in `_updateMultiVaultDepositPayload()`, `_updateSingleVaultDepositPayload()` and then the swapping can't be undone.
So if value is lost during swaps, bridging and depositing/redeeming this might not be detected and prevented.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _updateMultiVaultDepositPayload(
        ...
        if (PayloadUpdaterLib.validateSlippage(...)) {
            ... // update data
        } else {
            ... // set to failed
        }
    }
    function _updateSingleVaultDepositPayload(...) ... {
        ...
        if (PayloadUpdaterLib.validateSlippage(...)) {
            ... // update data
        } else {
           ... // set to failed
    }
    function _updateWithdrawPayload(...) ... {
        ...
        PayloadUpdaterLib.strictValidateSlippage(...); // reverts when slippage check fails
        ...
    }
    function _processMultiWithdrawal(...) ... {
        ...
        singleVaultData = InitSingleVaultData({
              ...
             maxSlippage: multiVaultData.maxSlippage[i],
             ...
        });
        ...
        try IBaseForm(superform_).xChainWithdrawFromVault(...) { // doesn't use maxSlippage
               ...
        } catch { ... }
    }
}
```

Recommendation: Implement slippage checks for swaps, bridging and depositing/redeeming on both the local chain and the xchain.
Superform: These are the slippage checks we do:

- We implement the slippage checks In xChainWithdrawFromVault.
- Since `singleVaultData_.amount` is the `super position` burned by the user and it represents the vault shares, the `dstAmount` could be the amount of collateral that we get for redeeming which in no way is

related to the other value for comparison.
- During withdrawals, there is an inherent slippage project when tx data is there, user tries to move x collateral/underlying they expect and if not there the withdrawals will fail automatically.
- When updating tx data by superform, there is slippage checks for the amount of collateral/underlying updated by superform using previewRedeem.

For the same chain withdrawal settlement case, we don't know what the user expects, so this case could be MEVed but direct checks against `singleVaultData.amount` won't hold good here.
Reviewer: Acknowledged

#M-22 : `_processDirectDeposit()` could leave dust

Context: ERC4626FormImplementation.sol#L108-L173
Description: Function `_processDirectDeposit()` checks that after a swap there are enough tokens (e.g. `singleVaultData_.amount`) and then it deposits this amount (e.g. `singleVaultData_.amount`). However if this swap results in more tokens than these extra tokens are not deposited and stay in the `form` contract as dust. If all the input parameters are set correctly the dust should be small, but with sudden changes in prices they may be larger. There is no way to retrieve these dust tokens from the `form`.
Also see issue: Function `ERC4626FormImplementation()` retrieved amount of tokens is different from swap input.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (ui
        vars.balanceBefore = IERC20(vars.collateral).balanceOf(address(this));
        ....
        if (singleVaultData_.liqData.txData.length > 0) {
            ...
            dispatchTokens(
                ...
                IBridgeValidator(vars.bridgeValidator).decodeAmountIn(singleVaultData_.liqData.txDat
                ...
            );
        }
        vars.balanceAfter = IERC20(vars.collateral).balanceOf(address(this));
        if (vars.balanceAfter - vars.balanceBefore < singleVaultData_.amount) {
            revert Error.DIRECT_DEPOSIT_INVALID_DATA();
        }
        ...
        IERC20(vars.collateral).safeIncreaseAllowance(vault, singleVaultData_.amount);
        dstAmount = v.deposit(singleVaultData_.amount, address(this));
    }
```

Recommendation: Consider to `deposit()` the entire output of the swap. Alternatively leave dust if it is small and transfer it to the Paymaster if it is larger.
Superform: Solved in PR 244.
Reviewer: Verified

#M-23 : `Forms` don't verify `vault` values

Context: ERC4626FormImplementation.sol#L35-L37
Description: `Forms` directly derive values from the underlying `vault` without verifying them Anybody can permissionlessly create Superforms by adding vaults to Forms, see Gitbook Superform v1, SuperformFactory, SuperformFactory. This means the vaults can't be trusted to always give back the same values.
At one moment `getVaultAsset()` of the form could return `tokenA` and another call is could return `tokenB`.
This is important when you want to check the tokens of a form, see issue:

- Functions `_processSingleDeposit()` and `_processMultiDeposit()` don't check the type of tokens

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function getVaultAsset() public view virtual override returns (address) {
```

```
        return address(IERC4626(vault).asset());
    }
```

Recommendation: Consider storing important values in the form, like the underlying asset.
Superform: Solved in PR 279.
Reviewer: Verified

#M-24 : No checks that `formBeaconId` corresponds to the `form`

Context: BaseForm.sol#L45-L54
Description: The `formBeaconIds` are used to check a form is paused. However there are no checks that the form corresponds to the `formBeaconId_` in the code. So by specifying a different `formBeaconId` that is not paused, its possible to circumvent the `pause` state.
The `formBeaconId_` isn't checked elsewhere either. This means there are multiple ways to deposits funds in the same `form`, with different `formBeaconIds`. This will result in different `superPositions` being minted, which is confusing and will make invariant checks more difficult to implement.

```
    modifier notPaused(InitSingleVaultData memory singleVaultData_) {
        (, uint32 formBeaconId_,) = singleVaultData_.superformId.getSuperform();
        if (
            IFormBeacon(
                ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTORY"))).getFormE
            ).paused() == 2
        ) revert Error.PAUSED();
        _;
    }
```

Recommendation: Check the `formBeaconId` corresponds to the `form`, this should be done for all deposits and withdraws, both local and on the `xchain`.
Superform: Solved in PR 227.
Reviewer: Verified

#M-25 : Forms are upgradable

Context: FormBeacon.sol#L42-L48, SuperformFactory.sol#L146-L156, BaseForm.sol#L20, ERC4626FormImplementation.sol#L17, ERC4626Form.sol#L10, ERC4626KYCDaoForm.sol#L13, ERC4626TimelockForm.sol#L18
Description: Forms are upgradable via the Beacon pattern. The creates a large centralisation (and thus rugpull) risk as the party controlling the beacon can also access the funds in the forms. The `form` contracts don't have `__gaps`, which makes upgrading them more difficult. As we have understood from the project the main goal for the Beacon pattern is to be able to pause all contract with the same implementation in one go.

```
contract SuperformFactory is ISuperformFactory {
    function updateFormBeaconLogic(uint32 formBeaconId_, address newFormLogic_) external override or
        ...
        FormBeacon(formBeacon[formBeaconId_]).update(newFormLogic_);
    }
}
```

```
import { UpgradeableBeacon } from "openzeppelin-contracts/contracts/proxy/beacon/UpgradeableBeacon.s
contract FormBeacon is IFormBeacon {
    function update(address formLogic_) external override onlySuperformFactory {
        beacon.upgradeTo(formLogic_);
        ...
    }
}
```

```
abstract contract BaseForm is Initializable, ERC165Upgradeable, IBaseForm { ... }
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler { ... }
contract ERC4626Form is ERC4626FormImplementation { ... }
contract ERC4626KYCDaoForm is ERC4626FormImplementation { ... }
contract ERC4626TimelockForm is ERC4626FormImplementation { ... }
```

Recommendation: Doublecheck the need for upgrability of the contract. Pausing several contracts with one transaction can also be achieved by letting the forms query to factory for their paused state. Forms can be deployed with clones with immutable arguments to save deployment costs while still being able to store a few parameters.
In you do want upgradeable contracts then consider doing the following:

- use storage gaps
- use the diamond storage pattern

Superform: Solved by PR 227 and PR 326.
Reviewer: Verified

#M-26 : Forms are not authenticated

Context: BaseRouterImplementation.sol#L572-L616, BaseRouterImplementation.sol#L669-L706, SuperformFactory.sol#L234-L241, DataLib.sol#L48-L56
Description: The functions `_directDeposit()` and `_directWithdraw()` seem to verify that a `superformId` exists on factory. However in practice the factory function `getSuperform()` calls the `DataLib` function `getSuperform()`.
Without this check any contract address could be supplied as part of a `superformId`, which could point to fake `form` addresses. Users that use these `forms` would probably loose their tokens. The lack of verification severly reduces the barriers for fraudsters to mislead users of the Superform protocol. Fake forms will also result in fake SuperPositions tokens and SuperTransmuter tokens, which reduced the trust in the SuperForms protocol. It also interferes with invariant checks.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _directDeposit(...) ... {
        /// @dev validates if superformId exists on factory
        (,, uint64 chainId) =  ISuperformFactory(...).getSuperform(superformId_);
    }
    function _directWithdraw(...) ... {
        /// @dev validates if superformId exists on factory
        (,, uint64 chainId) = ISuperformFactory().getSuperform(superformId_);
        ...
    }

contract SuperformFactory is ISuperformFactory {
    function getSuperform(uint256 superformId_) ... {
        (superform_, formBeaconId_, chainId_) = superformId_.getSuperform();
    }
}

library DataLib {
    function getSuperform(uint256 superformId_) ...  returns (address superform_, uint32 formBeaconI
        superform_ = address(uint160(superformId_));
        formBeaconId_ = uint32(superformId_ >> 160);
        chainId_ = uint64(superformId_ >> 192);
    }
}
```

Recommendation: Consider verifying the authenticy of the forms. If its decided not to check, then `SuperformFactory(...).getSuperform()` could be replaced with `DataLib .getSuperform()` to save some gas. If its deciced to check then `SuperformFactory(...).getSuperform()` should verify that the `form` address is one of the contracts it has deployed. The current datastructures of `SuperformFactory` don't allow efficient access to this information so they should be updated. On the `xchain` similar checks in `CoreStateRegistry` should be added to: `_processSingleDeposit()`, `_processMultiDeposit()`, `_processSingleWithdrawal()` and `_processMultiWithdrawal()`. A natural place to do this would be in `DataLib.validateSuperformChainId()`. For `TimelockStateRegistry` something similar should be done.
Superform: Solved in PR 297.
Reviewer: Verified

#M-27 : Local deposits and withdraws don't check the paused state of forms

Context: BaseRouterImplementation.sol#L182-L225, BaseRouterImplementation.sol#L309-L348, BaseRouterImplementation.sol#L381-L412, BaseRouterImplementation.sol#L381-L412, BaseRouterImplementation.sol#L712-L733

Description: The function `_singleDirectSingleVaultWithdraw()` uses `_buildWithdrawAmbData()` while that other comparable function don't use this. Function `_buildWithdrawAmbData()` is meant for bridge transaction so its not logical to use it. However the check `_validateSuperformData()` would be useful in all the functions. This function does the following checks:

- checks the form has the right chainId
- checks maxSlippage is within bounds
- checks the formBeacon isn't paused

Then chainId is also checked in `_directDeposit()` and `_directWithdraw()` of `BaseRouterImplementation`, so missing that check isn't important. `maxSlippage` isn't currently used for local deposit and withdraws. However the checking the pause is important.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle

    function _singleDirectSingleVaultDeposit(SingleDirectSingleVaultStateReq memory req_) internal v
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
        vars.currentPayloadId = ++payloadIds;
        ...
    }
    function _singleDirectMultiVaultDeposit(SingleDirectMultiVaultStateReq memory req_) internal vir
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
        vars.currentPayloadId = ++payloadIds;
        ...
    }
    function _singleDirectSingleVaultWithdraw(SingleDirectSingleVaultStateReq memory req_) internal
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
        ...
        (ambData, vars.currentPayloadId) = _buildWithdrawAmbData(msg.sender, vars.srcChainId, req_.s
        _directSingleWithdraw(ambData, msg.sender);
        ...
    }
    function _singleDirectMultiVaultWithdraw(SingleDirectMultiVaultStateReq memory req_) internal vi
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
        vars.currentPayloadId = ++payloadIds;
        ...
        IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnBatch(...);
        InitMultiVaultData memory vaultData = InitMultiVaultData(...);
        _directMultiWithdraw(vaultData, msg.sender);
    }
    function _buildWithdrawAmbData(... ) ... {
        ...
        if (!_validateSuperformData(dstChainId_, superformData_)) {
            revert Error.INVALID_SUPERFORMS_DATA();
        }
        IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnSingle(...);
        currentPayloadId = ++payloadIds;
        ...
    }
    function _validateSuperformData(...) ... {
        if (dstChainId_ != DataLib.getDestinationChain(superformData_.superformId)) return false;
        if (superformData_.maxSlippage > 10_000) return false;
        (, uint32 formBeaconId_,) = superformData_.superformId.getSuperform();
        return IFormBeacon(
            ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTORY"))).getFormBeacc
        ).paused() == 1;
```

```
    }
}
```

Recommendation: Make the code consistent, for example in the following way. Also see issue Inconsistent placement of `burn` functions.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle

    function _singleDirectSingleVaultDeposit(SingleDirectSingleVaultStateReq memory req_) internal v
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
+       if (!_validateSuperformData( vars.srcChainId, superformData_)) {
+           revert Error.INVALID_SUPERFORMS_DATA();
+       }
        vars.currentPayloadId = ++payloadIds;
        ...
    }
    function _singleDirectMultiVaultDeposit(SingleDirectMultiVaultStateReq memory req_) internal vir
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
+       if (!_validateSuperformsDepositData( ....)) {
+           revert Error.INVALID_SUPERFORMS_DATA();
+       }
        vars.currentPayloadId = ++payloadIds;
        ...
    }
    function _singleDirectSingleVaultWithdraw(SingleDirectSingleVaultStateReq memory req_) internal
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
+       if (!_validateSuperformData( vars.srcChainId, superformData_)) {
+           revert Error.INVALID_SUPERFORMS_DATA();
+       }
+       vars.currentPayloadId = ++payloadIds;
        ...
+       IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnSingle(...);
-       (ambData, vars.currentPayloadId) = _buildWithdrawAmbData(msg.sender, vars.srcChainId, req_.s
-       _directSingleWithdraw(ambData, msg.sender);
+       InitMultiVaultData memory vaultData = InitMultiVaultData(...);
+       _directMultiWithdraw(vaultData, msg.sender);
        ...
    }
    function _singleDirectMultiVaultWithdraw(SingleDirectMultiVaultStateReq memory req_) internal vi
        ActionLocalVars memory vars;
        vars.srcChainId = uint64(block.chainid);
+       if (!_validateSuperformsWithdrawData( ...)) {
+           revert Error.INVALID_SUPERFORMS_DATA();
+       }
        vars.currentPayloadId = ++payloadIds;
        ...
        IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnBatch(...);
        InitMultiVaultData memory vaultData = InitMultiVaultData(...);
        _directMultiWithdraw(vaultData, msg.sender);
    }
}
```

Superform: Solved by PR 298.
Reviewer: Verified

#M-28 : Function `_singleXChainSingleVaultDeposit` sets an allowance twice

Context: BaseRouterImplementation.sol#L127-L179, BaseRouterImplementation.sol#L855-L911, LiquidityHandler.sol#L24-L53, BaseRouterImplementation.sol#L424-L448
Description: Function `_singleXChainSingleVaultDeposit`, sets an allowance for the bridge twice:

1. via `_singleVaultTokenForward()`
2. via `_validateAndDispatchTokens()` and `dispatchToken()` from `LiquidityHandler`

The comparable function `_singleXChainMultiVaultDeposit()` supplies `new address[](0)` to `_multiVaultTokenForward()`, which results in not setting a (second) allowance.
Normally a higher allowance wouldn't be abused because a bridge would only transfer the number of tokens it is requested to transfer. However bugs in the bridge implemenation or in the superforms implementation might allow abuse.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _singleXChainSingleVaultDeposit(SingleXChainSingleVaultStateReq memory req_) internal v
        ...
       _singleVaultTokenForward(..., superRegistry.getBridgeAddress(vars.liqRequest.bridgeId), ... )
        ....
        _validateAndDispatchTokens(ValidateAndDispatchTokensArgs(...)); // second allowance
        ...
    }
    function _singleVaultTokenForward(..., address superform_, ... ) ... {
        if (vaultData_.liqData.token != NATIVE) {
            ...
            token.safeIncreaseAllowance(superform_, amount);
        }
    }
    function _validateAndDispatchTokens(ValidateAndDispatchTokensArgs memory args_) internal virtual
        ...
        dispatchTokens(...);
    }
}
```

```
abstract contract LiquidityHandler {
    function dispatchTokens(...) ... {
        if (token_ != NATIVE) {
            token.safeIncreaseAllowance(bridge_, amount_);
            ...
        } else {
            ...
        }
    }
}
```

Recommendation: To do the same as in `_singleXChainMultiVaultDeposit()` do something like this:

```
function _singleXChainSingleVaultDeposit(SingleXChainSingleVaultStateReq memory req_) internal virtu
    ...
-  _singleVaultTokenForward(..., superRegistry.getBridgeAddress(vars.liqRequest.bridgeId), ... ); //
+  _singleVaultTokenForward(..., address(0), ... );
    ....
}
function _singleVaultTokenForward(..., address superform_, ... ) ... {
     ...
+    if (superform_ != address(0))
         token.safeIncreaseAllowance(superform_ , amount);
}
```

Superform: Solved in PR 282.
Reviewer: Verified

Severity: Low Risk

#L-1 : Oracles could be stale

Context: PaymentHelper.sol#L740-L762
Description: The functions `_getGasPrice()` and `_getNativeTokenPrice()` make use of Chainlink's `latestRoundData` API, which returns the latest price data from a price feed.
It there is a problem with the oracle it might not be able to provide price data for a long period of time (longer than then normal heartbeat).

```
contract PaymentHelper is IPaymentHelper {
    function _getGasPrice(uint64 chainId_) internal view returns (uint256) {
      ...
      (, int256 value,, uint256 updatedAt,) = gasPriceOracle[chainId_].latestRoundData();
      ...
    }
    function _getNativeTokenPrice(uint64 chainId_) internal view returns (uint256) {
         ...
        (, int256 dstTokenPrice,, uint256 updatedAt,) = nativeFeedOracle[chainId_].latestRoundData()
        ...
    }
}
```

Recommendation: For each price feed oracle read the Chainlink documentation to understand the heartbeat (max update time), and configure the function `to check that theanswerreceived for a price feed was more recent than`PRICE_FEED_HEARTBEAT + 1 hour`, otherwise revert as the price is possibly stale.
Superform: Since `PaymentHelper` values are estimates for payments anyway, we'd actually rather have it return a stale number than revert. Some chains where we don't have a price feed will have hardcoded estimates (which could be stale as well, but we'll try and update them via a `keeper` on intervals).
Reviewer: Acknowledged

#L-2 : Uptime of L2 influences oracles

Context: PaymentHelper.sol#L740-L762
Description: The functions `_getGasPrice()` and `_getNativeTokenPrice()` make use of Chainlink's `latestRoundData` API, which returns the latest price data from a price feed.
As the protocol will be deployed to multiple EVM-compatible chains, one of which is Arbitrum for example. As you can see in the Chainlink docs this requires a check if the sequencer is currently up, and if it isn't the price shouldn't be used.

```
contract PaymentHelper is IPaymentHelper {
    function _getGasPrice(uint64 chainId_) internal view returns (uint256) {
      ...
      (, int256 value,, uint256 updatedAt,) = gasPriceOracle[chainId_].latestRoundData();
      ...
    }
    function _getNativeTokenPrice(uint64 chainId_) internal view returns (uint256) {
         ...
        (, int256 dstTokenPrice,, uint256 updatedAt,) = nativeFeedOracle[chainId_].latestRoundData()
        ...
    }
}
```

Recommendation: Follow the Chainlink docs here to add a sequencer check, but only in the cases where the protocol is deployed on an L2 - you can add a flag to indicate this.
Superform: We wouldn't want to revert payment estimates if the sequencer is down, stopping users from depositing or withdrawing to the chain. Not a time sensitive use case of oracles.
Reviewer: Acknowledged

#L-3 : Different proof sizes in `PaymentHelper`

Context: PaymentHelper.sol#L469-L537
Description: Function `_generateExtraData()` uses a proof size of 32, however function `_estimateAMBFees()` creates a proof which has a larger size. The version of `_estimateAMBFees()` seems to be the right one. An incorrect proof size could lead to sending insufficient payment for the transaction. Note: only `broadCast` uses a proof size of 32.

```
contract PaymentHelper is IPaymentHelper {
    function _generateExtraData(...) ... {
        ...
        uint256 totalDstGasReqInWeiForProof = 32 * gasPerKB[dstChainId_];
        ...
    }
```

```
    function _estimateAMBFees(...) ... {
        ...
        bytes memory proof_ = abi.encode(AMBMessage(type(uint256).max, abi.encode(keccak256(message_
        ...
    }
}
```

Recommendation: Doublecheck the correct broadcast size. Prefably use a constant for the size.
Superform: Solved in PR 240.
Reviewer: Verified

#L-4 : Irrelevant `sERC20`s can be deployed

Context: SuperTransmuter.sol#L92-L125, ERC1155A.sol#L366-L368
Description: The function `registerTransmuter()` allows creating a `sERC20()` even if there is no `superPosition` for the `superformId_`. It might be useful the first check this.
Also see issue Premissionless function `registerTransmuter()` can be abused.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(uint256 superformId_, bytes memory extraData_) external override ret
        ...
        synthethicTokenId[superformId_] = address(new sERC20(name,symbol,decimal));
        ...
    }
}
abstract contract ERC1155A is IERC1155A {
    function exists(uint256 id) external view virtual returns (bool) {
        return _totalSupply[id] > 0;
    }
}
```

Recommendation: Consider checking a `superPosition` exists for the `superformId_`. This can be done via the function `exists()`. Also consider making this function permissioned.
Superform: Solved in PR 313.
Reviewer: Verified

#L-5 : Constant instead of function selector

Context: LiFiTxDataExtractor.sol#L21-L45, StandardizedCallFacet.sol#L15
https://github.com/superform-xyz/superform-core/blob/2fa594b01e6c970200672a9b79018c11084032e6/src/crosschain-liquidity/lifi/LiFiValidator.sol#L243-L269
Description: The functions `_extractBridgeData()`, `_extractSwapData()` and `extractGenericSwapParameters()` use the constant `0xd6a4bc50`. This is difficult to reverse engineer, especially because two slightly different comments are made. Also the value changes if the function parameters would change. It is safer and more clear to the the function selector.

```
contract LiFiTxDataExtractor {
    function _extractBridgeData(bytes calldata data) internal pure returns (ILiFi.BridgeData memory
        if (abi.decode(data, (bytes4)) == 0xd6a4bc50) {
            // StandardizedCall
           ...
        }
        ...
    }
    function _extractSwapData(bytes calldata data) internal pure returns (LibSwap.SwapData[] memory
        if (abi.decode(data, (bytes4)) == 0xd6a4bc50) {
            // standardizedCall
            ...
        }
        ...
    }
}
```

```
contract LiFiValidator is BridgeValidator, LiFiTxDataExtractor {
    function extractGenericSwapParameters(bytes calldata data_) ... {
        ...
        if (abi.decode(data_, (bytes4)) == 0xd6a4bc50) {
            // standardizedCall
            ...
        }
    }
}

contract StandardizedCallFacet {
    function standardizedCall(bytes memory callData) external payable { ... }
}
```

Recommendation: Replace the constant with `standardizedCall.selector`.
Superform: Solved in PR 296.
Reviewer: Verified

#L-6 : Delay might not be set yet

Context: SuperRegistry.sol#L85-L94, CoreStateRegistry.sol#L312-L368, CoreStateRegistry.sol#L406-L408
Description: The delay may be set at a later moment via `setDelay()`. The functions `disputeRescueFailedDeposits()` and `finalizeRescueFailedDeposits()` don't verify the `delay > 0`. So the delay may be effectively 0.

```
contract SuperRegistry is ISuperRegistry, QuorumManager {
    function setDelay(uint256 delay_) external override onlyProtocolAdmin {
        ...
        delay = delay_;
        ...
    }
}
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function disputeRescueFailedDeposits(uint256 payloadId_) external override {
        ...
        if (... || block.timestamp > failedDeposits_.lastProposedTimestamp + _getDelay()) {
            revert ...;
        }
        ...
    }
    function finalizeRescueFailedDeposits(uint256 payloadId_) external override {
        ...
        if (... || block.timestamp < failedDeposits_.lastProposedTimestamp + _getDelay()) {
            revert ...;
        }
        ...
    }
    function _getDelay() internal view returns (uint256) {
        return superRegistry.delay();
    }
}
```

Recommendation: Consider checking that `delay` is set. This can also be done by creating a getter function `delay()` and checking it set there. Alternatively set the initial value for `delay` on a very high value.
Superform: Solved in PR 271.
Reviewer: Verified

#L-7 : Function `revokeRoleSuperBroadcast()` doesn't have a `nonce`

Context: SuperRBAC.sol#L78-L98, SuperformFactory.sol#L159-L183, SuperTransmuter.sol#L92-L125, WormholeSRImplementation.sol#L105-L133
Description: The functions `registerTransmuter()` and `changeFormBeaconPauseStatus()` have a `nonce` (`++xChainPayloadCounter`) when broadcast a message. However the comparable function `revokeRoleSuperBroadcast()` doesn't have this. This means when a `role/address` would be revoked twice

then the second message would be blocked `receiveMessage()`. Although its unlikely that this happens in practice, it is still unexpected behaviour.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(...) ... {
        ...
        BroadcastMessage memory transmuterPayload = BroadcastMessage(
            "SUPER_TRANSMUTER",
            DEPLOY_NEW_TRANSMUTER,
            abi.encode(uint64(block.chainid), ++xChainPayloadCounter, superformId_, name, symbol, de
        );
        _broadcast(abi.encode(transmuterPayload), extraData_);
        ...
    }
}
contract SuperformFactory is ISuperformFactory {
    function changeFormBeaconPauseStatus(...) ... {
        ...
        BroadcastMessage memory factoryPayload = BroadcastMessage(
            "SUPERFORM_FACTORY",
            SYNC_BEACON_STATUS,
            abi.encode(uint64(block.chainid), ++xChainPayloadCounter, formBeaconId_, paused_)
        );
        _broadcast(abi.encode(factoryPayload), extraData_);
        ...
    }
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    function revokeRoleSuperBroadcast(...) ... {
        BroadcastMessage memory rolesPayload = BroadcastMessage(
            "SUPER_RBAC",
            SYNC_REVOKE,
            abi.encode(role_, superRegistryAddressId_)
        );
        _broadcast(abi.encode(rolesPayload), extraData_);
        ...
    }
}
contract WormholeSRImplementation is IBroadcastAmbImplementation {
    function receiveMessage(bytes memory encodedMessage_) public {
        ...
        (IWormhole.VM memory wormholeMessage, bool valid,) = wormhole.parseAndVerifyVM(encodedMessag
        ...
        if (processedMessages[wormholeMessage.hash]) {
            revert Error.DUPLICATE_PAYLOAD();
        }
        processedMessages[wormholeMessage.hash] = true;
        ...
    }
}
```

Recommendation: Consider also having a `nonce` in `revokeRoleSuperBroadcast()`. An alternative way would be to move the `nonce` logic to `broadcastPayload()` of `WormholeSRImplementation`.
Superform: Solved in PR 251.
Reviewer: Verified

#L-8 : No check of `dstPayloadId_` / `payloadId_`

Context: PayloadHelper.sol#L86-L246 DstSwapper.sol#L70-L134, DstSwapper.sol#L160-L193
Description: Functions `decodeCoreStateRegistryPayload()` and `decodeCoreStateRegistryPayloadLiqData()` don't check `dstPayloadId_` so they could potentially try to access a message that hasn't been received. Function `processTx()` indirectly checks the validity of `payloadId_` via `_getFormUnderlyingFrom()` and `payloadTracking()`. This could be checked earlier, which would be more consistent with the rest of the code.

```
contract PayloadHelper is IPayloadHelper {
    function decodeCoreStateRegistryPayload(uint256 dstPayloadId_) ... {
        ... // no check of dstPayloadId_
```

```
    }
    function decodeCoreStateRegistryPayloadLiqData(uint256 dstPayloadId_) ... {
       ... // no check of dstPayloadId_
    }
}

contract DstSwapper is IDstSwapper, ReentrancyGuard {
    function processTx(uint256 payloadId_, ...) ... {
        ...
        v.underlying = _getFormUnderlyingFrom(payloadId_, index_);
        ...
     }
    function _getFormUnderlyingFrom(uint256 payloadId_, uint256 index_) internal view returns (addre
        ...
        PayloadState currState = coreStateRegistry.payloadTracking(payloadId_);
        ...
        if (currState != PayloadState.STORED) {
            revert Error.INVALID_PAYLOAD_STATUS();
        }
        ...
     }
}
```

Recommendation: Consider checking `dstPayloadId_` / `payloadId_` with `CoreStateRegistry.payloadsCount()`.
Superform: Solved in PR 274.
Reviewer: Verified

#L-9 : Non-initialization in for loop

Context: CoreStateRegistry.sol#L598, PayloadHelper.sol#L193
Description: Some of the `for` statements have as the first parameter a variable: `for (lV.i;....)`. When quickly scanning the code it seem like this initializes `lV.i` to `0`, however this is not the case. It is effectively a null statement. If multiple for loops are used in this way then `lV.i` continues with the value of the previous `for` loop. Currently this is no issue, but that could occur if code is copied in the future.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _updateWithdrawPayload(...) ... {
        ...
        for (lV.i; lV.i < lV.len;) {
            ...
       }
   }
}
contract PayloadHelper is IPayloadHelper {
    function decodeCoreStateRegistryPayloadLiqData(...) ... {
        ...
        for (v.i; v.i < len;) {
            ...
        }
    }
}
```

Recommendation: Do one of the following:

- remove the variable: `for (; lV.i < lV.len;)`
- assign a value: `for (lV.i = 0; lV.i < lV.len;)`

Superform: Solved in PR 300.
Reviewer: Verified

#L-10 : modifier `isValidPayloadId` missing

Context: CoreStateRegistry.sol#L64-L69, CoreStateRegistry.sol#L274-L368

Description: Most functions that interact with `payloadId` have a modifier `isValidPayloadId`, except from the following functions:

- `proposeRescueFailedDeposits()`
- `disputeRescueFailedDeposits()`
- `finalizeRescueFailedDeposits()`

Although other checks will prevent the functions from executing, its more consistent to have modifier `isValidPayloadId`. This will also be easier to debug failed transactions.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    modifier isValidPayloadId(uint256 payloadId_) {
        if (payloadId_ > payloadsCount) {
            revert Error.INVALID_PAYLOAD_ID();
        }
        _;
    }
    function proposeRescueFailedDeposits(uint256 payloadId_, ... ) ... {
    }
     function disputeRescueFailedDeposits(uint256 payloadId_) ... {
    }
    function finalizeRescueFailedDeposits(uint256 payloadId_) ... {
    }
}
```

Recommendation: Add the modifier `isValidPayloadId` to the functions:

- `proposeRescueFailedDeposits()`
- `disputeRescueFailedDeposits()`
- `finalizeRescueFailedDeposits()`
- **Superform:** Solved by PR 262.

Reviewer: Verified

#L-11 : Mint and burn asymmetric in `SuperTransmuter` / `Transmuter`

Context: Transmuter.sol#L67-L92
Description: When you directly mint `ERC20` via the `SuperTransmuter` they can't be turned into `1155A` NFTs because `transmuteToERC1155A()` doesn't have `ERC1155A` NFTs yet to return. The other way around it is no problem.
This also is complicated the with invariant calculations. For example:

- mint 100x 1155A NFT
- use `transmuteToERC20()`
- now you have 100x ERC20 + 100x 1155A in the Transmuter contract

So the invariant should be: `total amount erc20` + `total amount NFT` - `total amount NFT in Transmuter contract` == `total shares`.

```
abstract contract Transmuter is ITransmuter {
    function transmuteToERC1155A(uint256 id, uint256 amount) external override {
        sERC20 token = sERC20(synthethicTokenId[id]);
        token.burn(msg.sender, amount);
        ...
        ERC1155a.safeBatchTransferFrom(address(this), msg.sender, ids, amounts, bytes(""));
        ...
    }
    function transmuteToERC20(uint256 id, uint256 amount) external override {
        ERC1155a.safeTransferFrom(msg.sender, address(this), id, amount, "");
        sERC20(synthethicTokenId[id]).mint(msg.sender, amount);
        ...
    }
```

Recommendation: Consider doing one of the following:

- allow the SuperTransmuter to mint and burn ERC1155A tokens;
- integrate the SuperTransmuter and SuperPositions contracts and mint and burn both ERC20 and 1155A tokens. Add a signal to Deposit/Withdraw functions as well as the transported messages, which type of token is desired.

Superform: Solved in PR 313.
Reviewer: Verified

#L-12 : Premissionless function `registerTransmuter()` can be abused

Context: SuperTransmuter.sol#L92-L125
Description: The function `registerTransmuter()` is permissionless. This means it can be called or frontrun with other (not relevant) `extraData`. With empty `extraData` the `broadcast()` is skipped and can't be done again. When an inaccurate broadcast is detected, then a new form and a new transmuter can be registered. However the initial form can't function properly xchain because Transmuter tokens can't be minted when neccesary. Also see issues:

- No check `sERC20` has already been deployed;
- No recovery mechanism for return messages that can't be processed.

The initial form also can't be blocked.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(uint256 superformId_, bytes memory extraData_) external override ret
        ...
        /// @dev broadcast and deploy to the other destination chains
        if (extraData_.length > 0) {
            BroadcastMessage memory transmuterPayload = BroadcastMessage(
                "SUPER_TRANSMUTER",
                DEPLOY_NEW_TRANSMUTER,
                abi.encode(uint64(block.chainid), ++xChainPayloadCounter, superformId_, name, symbol
            );
            _broadcast(abi.encode(transmuterPayload), extraData_);
        }
        ...
    }
}
```

Recommendation: Make the `registerTransmuter()` permissioned.
Superform: Solved in PR 313.
Reviewer: Verified

#L-13 : `txHistory[]` not checked for validity

Context: SuperPositions.sol#L128-L219, SuperTransmuter.sol#L197-L295, DataLib.sol#L30-L41, PayloadHelper.sol#L249-L263
Description: The functions `stateMultiSync()` and `stateSync()` of `SuperPositions` and `SuperTransmuter` don't explictly check that `txHistory[...]` (`txInfo`) is valid. Although the following transaction probably will revert due to invalid values. For comparison function `decodeStateSyncerPayloadHistory()` does perform a check.

```
function stateMultiSync(AMBMessage memory data_)
    ...
    uint256 txInfo = txHistory[returnData.payloadId];
    ...
    (txType,,,, srcSender, srcChainId_) = txInfo.decodeTxInfo();
    ...
}
function stateSync(AMBMessage memory data_)
    ...
```

```
    uint256 txInfo = txHistory[returnData.payloadId];
    ...
   (txType,,,, srcSender, srcChainId_) = txInfo.decodeTxInfo();
    ...
}

function decodeTxInfo(uint256 txInfo_) ... {
    txType = uint8(txInfo_);
    callbackType = uint8(txInfo_ >> 8);
    multi = uint8(txInfo_ >> 16);
    registryId = uint8(txInfo_ >> 24);
    srcSender = address(uint160(txInfo_ >> 32));
    srcChainId = uint64(txInfo_ >> 192);
}

contract PayloadHelper is IPayloadHelper {
    function decodeStateSyncerPayloadHistory(...) ... {
        uint256 txInfo = IStateSyncer(superRegistry.getStateSyncer(superformRouterId_)).txHistory(sr
        if (txInfo != 0) {
            (txType, callbackType, multi,, srcSender, srcChainId) = txInfo.decodeTxInfo();
        }
    }
}
```

Recommendation: Consider checking `txInfo` is valid.
Superform: Solved by PR 261.
Reviewer: Verified

#L-14 : Use of `transferFrom()` and `transfer()`

Context: ERC4626FormImplementation.sol#L242-L264
https://github.com/superform-xyz/superform-core/blob/2fa594b01e6c970200672a9b79018c11084032e6/src/crosschain-data/extensions/CoreStateRegistry.sol#L340-L368
Description: The function `ERC4626FormImplementation()` uses `transferFrom()`. On other locations in the code `safeTransferFrom` is used. The function `finalizeRescueFailedDeposits()` uses `transfer()`. On other locations in the code `safeTransfer` is used.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
 function _processXChainDeposit(...) ... {
        ...
        IERC20(v.asset()).transferFrom(msg.sender, address(this), singleVaultData_.amount);
        /// @dev allowance is modified inside of the IERC20.transferFrom() call
        ...
    }
}
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function finalizeRescueFailedDeposits(uint256 payloadId_) external override {
        ...
        IERC20(IERC4626Form(form_).getVaultAsset()).transfer(refundAddress, amounts[i]);
        ...
    }
}
```

Recommendation: Consider using `safeTransferFrom()` and `safeTransfer`:

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
 function _processXChainDeposit(...) ... {
        ...
-       IERC20(v.asset()).transferFrom(msg.sender, address(this), singleVaultData_.amount);
+       IERC20(v.asset()).safeTransferFrom(msg.sender, address(this), singleVaultData_.amount);
-       /// @dev allowance is modified inside of the IERC20.transferFrom() call
+       /// @dev allowance is modified inside of the IERC20.safeTransferFrom() call
        ...
    }
}
```

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function finalizeRescueFailedDeposits(uint256 payloadId_) external override {
        ...
-       IERC20(IERC4626Form(form_).getVaultAsset()).transfer(refundAddress, amounts[i]);
+       IERC20(IERC4626Form(form_).getVaultAsset()).safeTransfer(refundAddress, amounts[i]);
        ...
    }
}
```

Superform: Solved in PR 215.
Reviewer: Verified

#L-15 : Tighter checks on `_processDirectDeposit()` and `_processDirectWithdraw()` `liqDstChainId()`

Context: ERC4626FormImplementation.sol#L108-L160, IBridgeValidator.sol#L10-L19
Description: The function `_processDirectDeposit()` specifies `singleVaultData_.liqData.liqDstChainId` for the `liqDstChainId`. As a same chain swap is done, it is also possible to use `vars.chainId`. This reduces the risk of accidentally sending the tokens across chain and also saves some gas.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (ui
        ...
        IBridgeValidator(vars.bridgeValidator).validateTxData(
            IBridgeValidator.ValidateTxDataArgs(
                ...
                vars.chainId, // srcChainId
                vars.chainId, // dstChainId
                singleVaultData_.liqData.liqDstChainId,  //liqDstChainId
                ...
            )
        );
       ...
    }
}
interface IBridgeValidator {
    struct ValidateTxDataArgs {
        ...
        uint64 srcChainId;
        uint64 dstChainId;
        uint64 liqDstChainId;
        ...
    }
}
```

Recommendation: Consider changing the code to:

```
IBridgeValidator(vars.bridgeValidator).validateTxData(
    IBridgeValidator.ValidateTxDataArgs(
        ...
        vars.chainId, // srcChainId
        vars.chainId, // dstChainId
-       singleVaultData_.liqData.liqDstChainId,
+       vars.chainId, //liqDstChainId
        ...
    )
);
```

Superform: Solved in PR 250.
Reviewer: Verified

#L-16 : Types of input and output tokens not checked in `ERC4626FormImplementation()`

Context: ERC4626FormImplementation.sol#L108-L173

Description: Function `ERC4626FormImplementation()` does not check the types of input and output tokens. The risk is limited though because the balance difference is checked. In order to create defensive code its safer to check the tokens.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (ui
        ...
        IERC4626 v = IERC4626(vault);
        vars.collateral = address(v.asset());
        ...
        IERC20 token = IERC20(singleVaultData_.liqData.token);
        ...
        token.safeTransferFrom(msg.sender, address(this), singleVaultData_.amount);
        if (singleVaultData_.liqData.txData.length > 0) {
             ... // swap
        }
        ...
        IERC20(vars.collateral).safeIncreaseAllowance(vault, singleVaultData_.amount);
        ...
        dstAmount = v.deposit(singleVaultData_.amount, address(this));
    }
```

Recommendation: Consider adding the following checks:

- when not swap is done: `singleVaultData_.liqData.token == vault.asset();`
- when a swap is done: output token of swap == `vault.asset()`.

Superform: Solved in PR 296.
Reviewer: Verified

#L-17 : `ERC4626FormImplementation()` retrieves `v.asset()` twice

Context: ERC4626FormImplementation.sol#L242-L264
Description: The function `ERC4626FormImplementation()` retrieve `v.asset()` twice. Some gas could be saved by storing this value in a temporary variable. The value `v.asset()` is derived from the `vault` and the `vault` could be malicious beause its added permissionlessly. This means that the second call could potentially return a different value. The risk is limited because the `form` only contains two potential tokens: the `underlying` asset and the `shares`. When a vault is malicious the `shares` are not valuable.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
 function _processXChainDeposit(...) ... {
        ...
        IERC20(v.asset()).transferFrom(msg.sender, address(this), singleVaultData_.amount);
        IERC20(v.asset()).safeIncreaseAllowance(vaultLoc, singleVaultData_.amount);
        ...
    }
```

Recommendation: Retrieve `v.asset()` only once and store it in a temporary value.
Superform: Solved in PR 263.
Reviewer: Verified

#L-18 : Constructor of `ERC4626FormImplementation()` doesn't check `stateRegistryId_`

Context: ERC4626FormImplementation.sol#L26-L28
Description: The constructor of `ERC4626FormImplementation()` doesn't check the `stateRegistryId_` is valid. An error in deployment could result in an invalid `form` being deployed. Its safer to check this.

```
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    constructor(address superRegistry_, uint256 stateRegistryId_) BaseForm(superRegistry_) {
        STATE_REGISTRY_ID = stateRegistryId_;
    }
```

Recommendation: Consider checking the `stateRegistryId_` is valid via a call to `getStateRegistry()`.

Superform: Solved in PR 260.
Reviewer:

#L-19 : `block.chainid` might not fit in an uint64

Context: SuperformRouter.sol#L34-L56, SuperformRouter.sol#L133-L157
Description: `block.chainid` is truncated to `uint64` to save spaces. In theory larger `block.chainids` could occur. Additonally in the functions `multiDstMultiVaultDeposit()` and `multiDstMultiVaultWithdraw()`, the result is stored in an `uint256`, while most other locations in the source use `uint64`.

```
contract SuperformRouter is BaseRouterImplementation {
    function multiDstMultiVaultDeposit(MultiDstMultiVaultStateReq calldata req_) ... {
        uint256 chainId = uint64(block.chainid);  // could be uint64
        ...
    }

    function multiDstMultiVaultWithdraw(MultiDstMultiVaultStateReq calldata req_) ... {
          uint256 chainId = uint64(block.chainid); // could be uint64
        ...
    }
}
```

Recommendation: Check the `block.chainid` fits in the maximum size of `uint64` in one or more constructors of key contracts.

```
if (block.chainid> type(uint64).max) {
    revert ...;
```

Also consider making this change.

```
contract SuperformRouter is BaseRouterImplementation {
    function multiDstMultiVaultDeposit(MultiDstMultiVaultStateReq calldata req_) ... {
-       uint256 chainId = uint64(block.chainid);
+       uint64 chainId = uint64(block.chainid);
        ...
    }

    function multiDstMultiVaultWithdraw(MultiDstMultiVaultStateReq calldata req_) ... {
-         uint256 chainId = uint64(block.chainid);
+         uint64 chainId = uint64(block.chainid);
        ...
    }
}
```

Superform: Solved in PR 258.
Reviewer: Verified

#L-20 : ChainId isn't checked for `0`

Context: SuperTransmuter.sol#L92-L125, DataLib.sol#L104-L111, BaseRouterImplementation.sol#L572-L598, BaseRouterImplementation.sol#L669-L706, BaseRouterImplementation.sol#L735-L782, BaseRouterImplementation.sol#L784-L834
Description: The `chainid` is regularly checked to confirm its valid. A value of `0` is also invalid, as can be seen in AMB adapters, however in the rest of the code this isn't checked. Checking with `0` also helps to detect invalid `superformIds`.
Its unlikely that `block.chainid== 0` but in theory it could be set to `0` or set to multiple of `2^64`.
Below are the situations where the chainid is checked, but where no 0 check occurs.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(uint256 superformId_, bytes memory extraData_) external override ret
        (address superform, uint32 formBeaconId, uint64 chainId) = DataLib.getSuperform(superformId_
        if (uint64(block.chainid) != chainId) revert Error.INVALID_CHAIN_ID();
        ...
    }
```

```
}
library DataLib {
    function validateSuperformChainId(uint256 superformId_, uint64 chainId_) internal pure {
        (,, uint64 chainId) = getSuperform(superformId_);
        if (chainId != chainId_) {
            revert Error.INVALID_CHAIN_ID();
        }
        ...
    }
}
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
     function _directDeposit(...) ... {
         (,, uint64 chainId) = ISuperformFactory(...).getSuperform(superformId_);
        ...
        if (chainId != uint64(block.chainid)) {
            revert Error.INVALID_CHAIN_ID();
        }
        ...
    }
    function _directWithdraw(...) ... {
        ...
        (,, uint64 chainId) = ISuperformFactory(....).getSuperform(superformId_);
        if (chainId != uint64(block.chainid)) {
            revert Error.INVALID_CHAIN_ID();
        }
        ...
    }
    function _validateSuperformsDepositData(...) ... {
        ...
        for (uint256 i; i < len;) {
            ...
            (, uint32 formBeaconId_, uint64 sfDstChainId) = superformsData_.superformIds[i].getSuper
            if (dstChainId_ != sfDstChainId) return false;
            ...
        }
    }
    function _validateSuperformsWithdrawData(...) ... {
        ...
        for (uint256 i; i < len;) {
             ...
            (, uint32 formBeaconId_, uint64 sfDstChainId) = superformsData_.superformIds[i].getSuper
            if (dstChainId_ != sfDstChainId) return false;
            ...
         }
    }
 }
```

Recommendation: Consider checking `chainid !=0` at the locations where the `chainid` is checked.
Superform: Solved in PR 269 and PR 318.
Reviewer: Verified

#L-21 : PERMIT2 might not be set

Context: SuperRegistry.sol#L97-L104, BaseRouterImplementation.sol#L56-L179, BaseRouterImplementation.sol#L855-L1003
Description: Several functions of `BaseRouterImplementation` use `superRegistry.PERMIT2()`. The function `setPermit2()` is used to set the value for `PERMIT2`. Depending on the deployment order, `PERMIT2` might potentially not be set. Additonally some chains might not support PERMIT2. When trying to call a function of `permit2`, a revert will occur however this might be difficult to debug.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    ...
    function _singleXChainMultiVaultDeposit(SingleXChainMultiVaultStateReq memory req_) internal vir
        ...
        address permit2 = superRegistry.PERMIT2();
        ...
```

```
    }
    function _singleXChainSingleVaultDeposit(SingleXChainSingleVaultStateReq memory req_) internal v
        ...
        _validateAndDispatchTokens(
            ValidateAndDispatchTokensArgs(
                vars.liqRequest, superRegistry.PERMIT2(), superform, vars.srcChainId, req_.dstChainI
            )
        );
        ...
    }
    function _singleVaultTokenForward(...) ... {
        ...
        address permit2 = superRegistry.PERMIT2();
        ...
    }
    function _multiVaultTokenForward(...) ... {
        ...
         v.permit2 = superRegistry.PERMIT2();
        ...
    }
}

contract SuperRegistry is ISuperRegistry, QuorumManager {
    function setPermit2(address permit2_) external override onlyProtocolAdmin {
        ...
        PERMIT2 = permit2_;
        ...
    }
}
```

Recommendation: Consider doing one of the following:

- check the value of `superRegistry.PERMIT2()` isn't `address(0)`;
- add a getter function that reverts if `PERMIT2` is `address(0)`. Use the getter instead of `superRegistry.PERMIT2()`.

Superform: Solved in PR 273.
Reviewer: Verified

Severity: Informational

#I-1 : `estimateFees()` ignore unsupported chains

Context: HyperlaneImplementation.sol#L181-L196, LayerzeroImplementation.sol#L313-L328, WormholeARImplementation.sol#L179-L201
Description: The `estimateFees()` of the AMB implementations return a value of `0` for `fees` if the `dstChainId_` isn't found. It might be more logical to revert.

```
contract HyperlaneImplementation is IAmbImplementation, IMessageRecipient {
    function estimateFees(...) ... {
        uint32 domain = ambChainId[dstChainId_];
        if (domain != 0) {
            fees = igp.quoteGasPayment(domain, abi.decode(extraData_, (uint256)));
        }
    }
}
```

Recommendation: Consider reverting if the chain isn't found.
Superform: Solved in PR 325.
Reviewer: Verified

#I-2 : Functions `_generateSingleVaultMessage()` and `_generateMultiVaultMessage()` use hardcoded value

Context: PaymentHelper.sol#L660-L703
Description: The functions `_generateSingleVaultMessage()` and `_generateMultiVaultMessage()` use a hardcoded value for `superformRouterId` of value `1`. This inflexible and not easy to read.

```
contract PaymentHelper is IPaymentHelper {
    function _generateSingleVaultMessage(SingleVaultSFData memory sfData_) ... {
        bytes memory ambData = abi.encode(
            InitSingleVaultData(
                1,
                ...
            )
        );
       ...
    }
    function _generateMultiVaultMessage(MultiVaultSFData memory sfData_) ... {
        bytes memory ambData = abi.encode(
            InitMultiVaultData(
                1,
                 ...
            )
        );
        ...
    }
}
```

Recommendation: Consider using a constant or provide the value via the constructor.
Superform: Resolved while fixing other issues.
Reviewer: Verified

#I-3 : Address `DST_SWAPPER` retrieved inside a loop

Context: PaymentHelper.sol#L588-L623
Description: Function `_estimateSwapFees()` uses `superRegistry.getAddress(keccak256("DST_SWAPPER"))` inside a loop. It never changes so could be retrieved outside the loop. This code also assumes the address for `DST_SWAPPER` is the same on everychain.

```
contract PaymentHelper is IPaymentHelper {
    function _estimateSwapFees(
        for (uint256 i; i < liqReq_.length;) {
          ...
          ...   superRegistry.getAddress(keccak256("DST_SWAPPER")) ...
          ...
    }
}
```

Recommendation: Doublecheck the assumption that the address of `DST_SWAPPER` is the same on every chain and they will also stay the same. If not: then change the code. If they stay the same, retrieve `superRegistry.getAddress(keccak256("`superRegistry.getAddress(keccak256("DST_SWAPPER")) `inside a loop"))` outside the loop.
Superform: `DstSwapper` might not be the same on all chains, but `hasDstSwap` can be used. Solved in PR 310.
Reviewer: Verified

#I-4 : `AmbIds` in `_generateExtraData` hardcoded

Context: PaymentHelper.sol#L469-L503, Abstract.Deploy.s.sol#L157-L163
Description: The `ambIds` in function `_generateExtraData()` are hardcoded which is inflexible. The `amdId` for `broadcasts` (`wormhole SR`) isn't present, although it doesn't seem to be used.

```
contract PaymentHelper is IPaymentHelper {
    function _generateExtraData(...) ... {
        ...
            if (ambIds_[i] == 1) {
                extraDataPerAMB[i] = abi.encodePacked(uint16(2), gasReq, uint256(0), address(0));
```

```
            }
            if (ambIds_[i] == 2) {
                extraDataPerAMB[i] = abi.encode(gasReq);
            }
            if (ambIds_[i] == 3) {
                extraDataPerAMB[i] = abi.encode(0, gasReq);
            }
        ...
    }
}

/// @notice id 1 is layerzero
/// @notice id 2 is hyperlane
/// @notice id 3 is wormhole AR
/// @notice 4 is wormhole SR
uint8[] public ambIds = [uint8(1), 2, 3, 4];
bool[] public broadcastAMB = [false, false, false, true];
```

Recommendation: Use constants and/or comments to explain the numbers. Consider retrieving the data from the `AMB` implementation, for example something like:

```
IAmbImplementation(superRegistry.getAmbAddress(ambIds_[i])).generateExtraData(gasReq)
```

Doublecheck the need to add a case for `broadcast ambids`.
Superform: Solved by PR 336.
Reviewer: Verified

#I-5 : Is gas per byte or per kilobyte?

Context: PaymentHelper.sol#L469-L503
Description: Function `_generateExtraData()` multiplies a `bytes` length with `gasPerKB[]`. There doesn't seem to be scaling from `kilobyte` to `byte`. This doesn't seem logical.

```
contract PaymentHelper is IPaymentHelper {
    function _generateExtraData(...) ... {
        ...
        uint256 totalDstGasReqInWei = message_.length * gasPerKB[dstChainId_];
        uint256 totalDstGasReqInWeiForProof = 32 * gasPerKB[dstChainId_];
        ...
    }
}
```

Recommendation: Doublecheck the goal. If the gas is per kilobyte, then the message lengths have to be scaled down to kilobytes. If the gas is per byte the consider changing the name.
Superform: Solved by PR 336.
Reviewer: Verified

#I-6 : Public function names start with _

Context: PaymentHelper.sol#L469-L570
Description: The function names of `_generateExtraData()`, `_estimateAMBFees()` and `_estimateAMBFeesReturnExtraData()` start with an _, while they are `public`. Normally only internal functions start with a _. Currently it is not consistent.

```
contract PaymentHelper is IPaymentHelper {
    function _generateExtraData(...) public  view ... {
        ...
    }
    function _estimateAMBFees(...) public  view ... {
        ...
    }
    function _estimateAMBFeesReturnExtraData(...) public  view ... {
        ...
```

```
    }
}
```

Recommendation: Either change the functions to `internal` or remove the leading _.
Superform: Resolved while fixing other issues.
Reviewer: Verified

#I-7 : `TIMELOCK_FORM_ID` not used for xchain estimate functions

Context: PaymentHelper.sol#L380-L432
Description: Only the functions `estimateSingleDirectSingleVault()` and `estimateSingleDirectMultiVault()` take in account `TIMELOCK_FORM_ID`. However for the xchain withdraw `TIMELOCK_FORM_ID` could also be relevant.

```
contract PaymentHelper is IPaymentHelper {
    function estimateSingleDirectSingleVault(...) ... {
        ...
        if (!isDeposit_ && formId == TIMELOCK_FORM_ID) {
            srcAmount += twoStepCost[uint64(block.chainid)] * _getGasPrice(uint64(block.chainid));
        }
        ...
    }
    function estimateSingleDirectMultiVault(...) ... {
        ...
        if (!isDeposit_ && formId == TIMELOCK_FORM_ID) {
            srcAmount += twoStepCost[uint64(block.chainid)] * _getGasPrice(uint64(block.chainid));
        }
        ...
    }
}
```

Recommendation: Doublecheck the need to add `TIMELOCK_FORM_ID` for xchain functions.
Superform: Solved by PR 336.
Reviewer: Verified

#I-8 : functions `estimateMultiDstMultiVault()` only adds amount in the deposit case

Context: PaymentHelper.sol#L196-L247
Description: The functions `estimateMultiDstMultiVault()` only adds amount in the deposit case. However (as also noted in the comment), in the withdraw case, sometimes a return message is sent too. Not adding this means the protocol has to pay for these messages. Note: several other function do the same.

```
contract PaymentHelper is IPaymentHelper {
    function estimateMultiDstMultiVault(...) ... {
        ...
        if (isDeposit_) {
            /// @dev step 3: estimation processing cost of acknowledgement
            /// @notice optimistically estimating. (Ideal case scenario: no failed deposits / withdr
            srcAmount += _estimateAckProcessingCost(req_.dstChainIds.length, superformIdsLen);
        }
        ...
    }
}
```

Recommendation: Consider adding a mechanism to be able to add costs for withdraw return messages in case these cost could not be trivial.
Superform: Optimistically, there is no need for acknowledgement in case of withdrawals, hence its not added to the general estimation. In-case of withdrawal failure (which is a pessimistic case) then use might top-up their gas fees to paymaster. But ideally that's not the fair path, hence not included in the estimations.
Reviewer: Acknowledged

#I-9 : Prevent mistakes with `totalDstGas`

Context: PaymentHelper.sol#L196-L247
Description: Variable `totalDstGas` has to start at value 0 at every iteration of the for loop. If someone would move `uint256 totalDstGas;` outside of the for loop, for example to try and save gas, the assignment wouldn't be done and the function result would be inaccurate.

```
contract PaymentHelper is IPaymentHelper {
    function estimateMultiDstMultiVault(...) ... {
        for (uint256 i; i < len;) {
            uint256 totalDstGas;
            ... // update totalDstGas
            dstAmount += _convertToNativeFee(req_.dstChainIds[i], totalDstGas);
            ...
        }
        totalAmount = srcAmount + dstAmount + liqAmount;
    }
}
```

Recommendation: For defensive programming it is be better to assign `totalDstGas = 0` in the for loop. If stack size permits then `uint256 totalDstGas;` can be moved outside the for loop.
Superform: Solved by PR 336.
Reviewer: Verified

#I-10 : Both functions `addChain()` and `updateChainConfig()` can do the same

Context: PaymentHelper.sol#L85-L171
Description: Both functions `addChain()` and `updateChainConfig()` allow adding chains as well as updating values. They do have different authorization modifiers but they can basically do the same. However the function names indicate a differerence.

```
contract PaymentHelper is IPaymentHelper {
    function addChain(...) ... onlyProtocolAdmin  {
        ...
        nativeFeedOracle[chainId_] = ...
        ... // more assignments
        swapGasUsed[chainId_] = ...
        ...
    }
}
function updateChainConfig(...) ... onlyEmergencyAdmin {
    ...
    nativeFeedOracle[chainId_] = ...
    ... // more assignments
    swapGasUsed[chainId_] = ...
    ...
}
```

Recommendation: Double check the goals of the functions. Either adapt the functionality to the function name, or adapt the function name to the implementation.
Superform: Changed the function names to make it more clear in PR 322.
Reviewer: Verified

#I-11 : Code duplication between `Transmuter` and `ERC1155TokenReceiver`

Context: Transmuter.sol#L103-L129, ERC1155A.sol#L490-L507
Description: The functions `onERC1155Received()` and `onERC1155BatchReceived()` of `Transmuter` are equivalent to the functions in `ERC1155TokenReceiver` of `ERC1155A.sol`.

```
abstract contract Transmuter is ITransmuter {
    function onERC1155Received(...) ... {
        return this.onERC1155Received.selector;
    }
    function onERC1155BatchReceived(...) ... {
```

```
        return this.onERC1155BatchReceived.selector;
    }

abstract contract ERC1155TokenReceiver {
    function onERC1155Received(...) ... {
        return ERC1155TokenReceiver.onERC1155Received.selector;
    }
    function onERC1155BatchReceived(...) ... {
        return ERC1155TokenReceiver.onERC1155BatchReceived.selector;
    }
```

Recommendation: Consider inhereting from `ERC1155TokenReceiver`.
Superform: Solved by integrating Transmuter integrated into ERC1155A.
Reviewer: Verified

#I-12 : `synthethicTokenId[]== 0` not checked

Context: Transmuter.sol#L51-L97
Description: The functions `transmuteBatchToERC20()`, `transmuteToERC20()` and `transmuteToERC1155A()` don't check if `synthethicTokenId[]== 0`. If its `0` then the call to this address will fail, but this might be difficult to debug.

```
abstract contract Transmuter is ITransmuter {
    function transmuteBatchToERC20(uint256[] memory ids, uint256[] memory amounts) external override
        ...
        for (uint256 i = 0; i < ids.length; i++) {
            sERC20(synthethicTokenId[ids[i]]).mint(...);
        }
        ...
    }
    function transmuteToERC20(uint256 id, uint256 amount) external override {
        ...
        sERC20(synthethicTokenId[id]).mint(...);
        ...
    }
    function transmuteToERC1155A(uint256 id, uint256 amount) external override {
        sERC20 token = sERC20(synthethicTokenId[id]);
        token.burn(msg.sender, amount);
        ...
    }
}
```

Recommendation: Consider detecting `synthethicTokenId[]== 0` and the revert with an appropriate error message.
Superform: Solved in ERC1155A PR 23https://github.com/superform-xyz/ERC1155A/pull/23).
Reviewer: Verified

#I-13 : Use of `bridge` versus `ambId` is confusing

Context: SuperRegistry.sol#L143-L196
Description: The `SuperRegistry` contains mappings for `bridges` and `ambIds`. As an AMB is also a bridge, this is confusing, especially because in the `amb` mappings, the name `bridgeId` is used too.

```
contract SuperRegistry is ISuperRegistry, QuorumManager {
    ...
    mapping(uint8 bridgeId => address bridgeAddress) public bridgeAddresses;
    mapping(uint8 bridgeId => address bridgeValidator) public bridgeValidator;
    ...
    mapping(uint8 bridgeId => address ambAddresses) public ambAddresses; // ambId
    mapping(uint8 bridgeId => bool isBroadcastAMB) public isBroadcastAMB; // ambId
    mapping(address ambAddress => uint8 bridgeId) public ambIds; // ambId
    ...
    function setAmbAddress(
        ...
        ambAddresses[ambId] = ambAddress;
```

```
        ambIds[ambAddress] = ambId;
        isBroadcastAMB[ambId] = broadcastAMB;
        ...
    }
}
```

Recommendation: Consider changing the code to:

```
-mapping(uint8 bridgeId => address ambAddresses) public ambAddresses;
+mapping(uint8 ambId=> address ambAddresses) public ambAddresses;
-mapping(uint8 bridgeId => bool isBroadcastAMB) public isBroadcastAMB;
+mapping(uint8 ambId=> bool isBroadcastAMB) public isBroadcastAMB;
-mapping(address ambAddress => uint8 bridgeId) public ambIds;
+mapping(address ambAddress => uint8 ambId) public ambIds;
```

And for the remaining occurances of `bridge`, use something like `liquidityBridge`

```
-bridge...
-liquidityBridge...
```

Superform: Solved in PR 330.
Reviewer: Verified

#I-14 : Similar functions `hasProtocolAdminRole()` and `hasEmergencyAdminRole()` have different checks

Context: https://github.com/superform-xyz/superform-core/blob/2fa594b01e6c970200672a9b79018c11084032e6/src/settings/SuperRBAC.sol#L124-L132
Description: The functions `hasProtocolAdminRole()` and `hasEmergencyAdminRole()` are very similar however hasProtocolAdminRole()`has an extra check. The check foraddress(0doesn't seem neccesary becausehasRole()doesn't have an exception foraddress(0)although it does have an exception forrole 0`.

```
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    function hasProtocolAdminRole(address admin_) external view override returns (bool) {
        if (admin_ == address(0)) return false;
        return hasRole(PROTOCOL_ADMIN_ROLE, admin_);
    }
    function hasEmergencyAdminRole(address emergencyAdmin_) external view override returns (bool) {
        return hasRole(EMERGENCY_ADMIN_ROLE, emergencyAdmin_);
    }
}
```

Recommendation: Make the functions `hasProtocolAdminRole()` and `hasEmergencyAdminRole()` consistent.
Superform: Solved in PR 330.
Reviewer: Verified

#I-15 : `revokeRoleSuperBroadcast()` and `stateSyncBroadcast()` derive address to revoke in different way

Context: SuperRBAC.sol#L78-L117
Description: Function `revokeRoleSuperBroadcast()` revokes the role of `addressToRevoke_` while its mirror function on the xchains `stateSyncBroadcast()`, uses `getAddress(superRegistryAddressId)` to revoke the role. It seems they could both use `getAddress(superRegistryAddressId)` for consistency.

```
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    function revokeRoleSuperBroadcast(..., address addressToRevoke_, ..., bytes32 superRegistryAddre
        ...
        revokeRole(role_, addressToRevoke_);
         ...
    }
    function stateSyncBroadcast(bytes memory data_) external override {
        ...
        address addressToRevoke = superRegistry.getAddress(superRegistryAddressId);
```

```
        ...
        _revokeRole(role, addressToRevoke);
        }
    }
}
```

Recommendation: Let function `revokeRoleSuperBroadcast()` also use `getAddress(superRegistryAddressId)`, after doublechecking this is a sound approach.
Superform: Solved in PR 305.
Reviewer: Verified

#I-16 : Role configuration is very important

Context: SuperRBAC.sol#L14-L75
Description: There are quite a lot of roles and chains. If a mistake is made with the configuration of the roles then tokens can be stolen and invariant can be breached.

```
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    bytes32 public constant override PROTOCOL_ADMIN_ROLE =
    bytes32 public constant override EMERGENCY_ADMIN_ROLE =
    bytes32 public constant override PAYMENT_ADMIN_ROLE =
    bytes32 public constant override BROADCASTER_ROLE =
    bytes32 public constant override CORE_STATE_REGISTRY_PROCESSOR_ROLE =
    bytes32 public constant override TIMELOCK_STATE_REGISTRY_PROCESSOR_ROLE =
    bytes32 public constant override BROADCAST_STATE_REGISTRY_PROCESSOR_ROLE =
    bytes32 public constant override CORE_STATE_REGISTRY_UPDATER_ROLE =
    bytes32 public constant override CORE_STATE_REGISTRY_RESCUER_ROLE =
    bytes32 public constant override CORE_STATE_REGISTRY_DISPUTER_ROLE =
    bytes32 public constant override SUPERPOSITIONS_MINTER_ROLE =
    bytes32 public constant override SUPERPOSITIONS_BURNER_ROLE =
    bytes32 public constant override SERC20_MINTER_ROLE =
    bytes32 public constant override SERC20_BURNER_ROLE =
    bytes32 public constant override MINTER_STATE_REGISTRY_ROLE =
    bytes32 public constant override WORMHOLE_VAA_RELAYER_ROLE =
    bytes32 public constant override DST_SWAPPER_ROLE =

    function setRoleAdmin(bytes32 role_, bytes32 adminRole_) external override onlyRole(PROTOCOL_ADM
        _setRoleAdmin(role_, adminRole_);
    }
}
```

Recommendation: Setup a system to carefully manage the configuration and the changes to it. Also have a monitoring mechanism to verify the correct setup.
Superform: We are enabling a monitoring system via usage of Hypernative and Openzeppelin Defender. We also plan to add assertions at the deployment script level and ensure those assertions pass first in an automated Tenderly devotes deployment.
Reviewer: Acknowledged

#I-17 : Function `validateDepositPayloadUpdate()` and `validateWithdrawPayloadUpdate()` are similiar

Context: PayloadUpdaterLib.sol#L67-L111
Description: Function `validateDepositPayloadUpdate()` and `validateWithdrawPayloadUpdate()` are similiar. The only difference is the comparison to `DEPOSIT` versus `WITHDRAW`. This value could be supplied as a parameter, comparable to `isMulti_` and then the functions could be integrated.

```
library PayloadUpdaterLib {
    function validateDepositPayloadUpdate(...) ... {
        (uint256 txType, uint256 callbackType, uint8 multi,,,) = DataLib.decodeTxInfo(txInfo_);
        if (txType != uint256(TransactionType.DEPOSIT) || callbackType != uint256(CallbackType.INIT)
        if (currentPayloadState_ != PayloadState.STORED) { revert ... }
        if (multi != isMulti_) { revert ... }
    }
```

```
    function validateWithdrawPayloadUpdate(...) ... {
        (uint256 txType, uint256 callbackType, uint8 multi,,,) = DataLib.decodeTxInfo(txInfo_);
        if (txType != uint256(TransactionType.WITHDRAW) || callbackType != uint256(CallbackType.INIT
        if (currentPayloadState_ != PayloadState.STORED) { revert ... }
        if (multi != isMulti_) { revert ... }
    }
}
```

Recommendation: Consider integrating the functions `validateDepositPayloadUpdate()` and `validateWithdrawPayloadUpdate()`.
Superform: Solved in PR 333.
Reviewer: Verified

#I-18 : Function `packTxInfo()` can be changed to Solidity

Context: DataLib.sol#L7-L41
Description: Function `packTxInfo()` is written in assembly, while the reverse function `decodeTxInfo` is written in Solidity. Changing `packTxInfo()` is easier for reviewing and mainting the code.

```
library DataLib {
    function packTxInfo(...) ... {
        assembly ("memory-safe") {
            txInfo := txType_
            txInfo := or(txInfo, shl(8, callbackType_))
            txInfo := or(txInfo, shl(16, multi_))
            txInfo := or(txInfo, shl(24, registryId_))
            txInfo := or(txInfo, shl(32, srcSender_))
            txInfo := or(txInfo, shl(192, srcChainId_))
        }
    }
    function decodeTxInfo(uint256 txInfo_) ... {
        txType = uint8(txInfo_);
        callbackType = uint8(txInfo_ >> 8);
        multi = uint8(txInfo_ >> 16);
        registryId = uint8(txInfo_ >> 24);
        srcSender = address(uint160(txInfo_ >> 32));
        srcChainId = uint64(txInfo_ >> 192);
    }
}
```

Recommendation: Consider changing the code to: Note: this uses a very small amount of extra gas.

```
function packTxInfo(...) ... {
    txInfo = txType_
        | uint256(callbackType_) << 8
        | uint256(multi_) << 16
        | uint256(registryId_) << 24
        | uint256(uint160(srcSender_)) << 32
        | uint256(srcChainId_) << 192;
}
```

Superform: Solved in PR 237.
Reviewer: Verified

#I-19 : Combine two almost identical calls in `dispatchTokens()`

Context: LiquidityHandler.sol#L24-L53
Description: The code to call an external contract is present in two instances in `dispatchTokens()`. The only difference is the error message. As these pieces of code are important and error phrone, it might be better to integrate them. This will also make maintenance easier.

```
abstract contract LiquidityHandler {
     function dispatchTokens(...) ... {
         ...
         if (token_ != NATIVE) {
```

```
            ...
            (bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
            if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA();
            ...
        } else {
            ...
            (bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
            if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA_NATIVE();
            ...
        }
        ...
    }
}
```

Recommendation: Consider integrating the code to the following: Note: the parameter to `FAILED_TO_EXECUTE_TXDATA(...)` allows the offchain indexers to differentiate the cases.

```
(bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA(token_);
```

Superform: Solved in PR 305.
Reviewer: Verified

#I-20 : `internal` function `dispatchTokens()` name doesn't start with an '_'.

Context: LiquidityHandler.sol#L24-L53
Description: Most `internal` function names start with an *'. Function `dispatchTokens()` is an exception to this. Note: internal library function don't start with an "*.

```
abstract contract LiquidityHandler {
    function dispatchTokens(...) internal ... {
        ...
    }
}
```

Recommendation: Consider starting the function name with an _.

```
-function dispatchTokens(...) internal ... {
+function _dispatchTokens(...) internal ... {
   ...
}
```

Superform: Solved in PR 305.
Reviewer: Verified

#I-21 : `LiquidityHandler` has two functions

Context: LiquidityHandler.sol#L11
Description: A comment in `LiquidityHandler` isn't accurate because the contract is also used to swap tokens. Also the name function `dispatchTokens()` doesn't indicate it can be use to swap tokens.

```
* @dev bridges tokens from Chain A -> Chain B. To be inherited by contracts that move liquidity
abstract contract LiquidityHandler {
    function dispatchTokens(...) ... {
        ...
    }
}
```

Recommendation: Consider to update the name `dispatchTokens()` to something that includes the swapping function. Consider changing the comment to:

```
-* @dev bridges tokens from Chain A -> Chain B. To be inherited by contracts that move liquidity
+* @dev bridges tokens from Chain A -> Chain B or swaps tokens.
+* @dev To be inherited by contracts that move liquidity or do swaps
```

Superform: Solved in PR 305.
Reviewer: Verified

#I-22 : Function `processTx()` can call `dispatchTokens()`

Context: DstSwapper.sol#L70-L134, LiquidityHandler.sol#L24-L53
Description: Part of the code of `processTx()` is very similar to the code of `dispatchTokens()`. Function `processTx()` could call `dispatchTokens()` for easier maintenance.

```
contract DstSwapper is IDstSwapper, ReentrancyGuard {
    function processTx(...) ... {
        ...
        if (approvalToken_ != NATIVE) {
            IERC20(approvalToken_).safeIncreaseAllowance(v.to, amount_);
            (bool success,) = payable(v.to).call(txData_);
            if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA();
        } else {
            (bool success,) = payable(v.to).call{ value: amount_ }(txData_);
            if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA_NATIVE();
        }
        ...
    }
abstract contract LiquidityHandler {
    function dispatchTokens(...) ... {
        if (token_ != NATIVE) {
            IERC20 token = IERC20(token_);
            token.safeIncreaseAllowance(bridge_, amount_);
            unchecked {
                (bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
                if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA();
            }
        } else {
            if (nativeAmount_ < amount_) revert Error.INSUFFICIENT_NATIVE_AMOUNT();
            unchecked {
                (bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
                if (!success) revert Error.FAILED_TO_EXECUTE_TXDATA_NATIVE();
            }
        }
    }
}
```

Recommendation: Let function `processTx()` call `dispatchTokens()`. Doublecheck and integrate the small differences.
Superform: Solved in PR 305.
Reviewer: Verified

#I-23 : Incorrect/unexpected messages are return empty values

Context: PayloadHelper.sol#L86-L160, PayloadHelper.sol#L279-L301
Description: In functions `decodeCoreStateRegistryPayload()` and `decodeTimeLockFailedPayload()`: if the `callbackType_` is not recognized then empty values are returned. The functions could also revert in that situation. Also see issue: Incorrect/unexpected messages are ignored.

```
contract PayloadHelper is IPayloadHelper {
    function decodeCoreStateRegistryPayload(uint256 dstPayloadId_) ... {
        if (v.callbackType == uint256(CallbackType.RETURN) || v.callbackType == uint256(CallbackType
           ... // set values
        }
        if (v.callbackType == uint256(CallbackType.INIT)) {
           ... // set values
        }
        return (...);
    }
    function decodeTimeLockFailedPayload(uint256 timelockPayloadId_)
        if (callbackType_ == uint256(CallbackType.FAIL)) {
```

```
            ... // set values
        }
    }
}
```

Recommendation: Consider to revert when an incorrect/unexpected message is received.
Superform: Solved in PR 335
Reviewer: Verified

#I-24 : No check of `timelockPayloadId_`

Context: PayloadHelper.sol#L266-L301
Description: Functions `decodeCoreStateRegistryPayload()` and `decodeCoreStateRegistryPayloadLiqData()` don't check `timelockPayloadId_` so they could potentially try to access a message that hasn't been received.

```
contract PayloadHelper is IPayloadHelper {
    function decodeTimeLockPayload(uint256 timelockPayloadId_) ... {
        ... // no check of timelockPayloadId_
    }
    function decodeTimeLockFailedPayload(uint256 timelockPayloadId_) ... {
        ... // no check of timelockPayloadId_
    }
}
```

Recommendation: Consider checking `timelockPayloadId_` with `TimelockStateRegistry.timelockPayloadCounter()`.
Superform: Solved by PR 335.
Reviewer: Verified

#I-25 : `decodeStateSyncerPayloadHistory()` could revert on invalid data

Context: PayloadHelper.sol#L249-L263
Description: When calling `decodeStateSyncerPayloadHistory()` it is not straightforward to know if the result is invalid.

```
contract PayloadHelper is IPayloadHelper {
    function decodeStateSyncerPayloadHistory(...) ... {
        uint256 txInfo = IStateSyncer(superRegistry.getStateSyncer(superformRouterId_)).txHistory(sr
        if (txInfo != 0) {
            (txType, callbackType, multi,, srcSender, srcChainId) = txInfo.decodeTxInfo();
        }
    }
}
```

Recommendation: Consider revering when `txInfo == 0`.
Superform: Solved in PR 335.
Reviewer: Verified

#I-26 : Function `decodeCoreStateRegistryPayload()` doesn't return all available data

Context: PayloadHelper.sol#L86-L160
Description: There are more fields that could be returned by `decodeCoreStateRegistryPayload()`.

```
contract PayloadHelper is IPayloadHelper {
    function decodeCoreStateRegistryPayload(uint256 dstPayloadId_) ... {
        ...
        return (
            v.txType,
            v.callbackType,
            v.srcSender,
            v.srcChainId,
            v.amounts,
```

```
            v.slippage,
            v.superformIds,
            v.srcPayloadId,
            v.superformRouterId
        );
    }
}
```

Recommendation: Doublecheck if any of the following fields are also useful to return:

- `hasDstSwaps`
- `liqData`
- `dstRefundAddress`
- `extraFormData`

Superform: Solved in PR 335.
Reviewer: Verified

#I-27 : Validity check of `timeLockPayloadId_` in function `finalizePayload()`

Context: TimelockStateRegistry.sol#L109-L183
Description: If an invalid value for `timeLockPayloadId_` is (accidentally) passed to `finalizePayload`, it will be detected by the check `p.status != TwoStepsStatus.PENDING`. However before that a call is already done to `getBridgeValidator()`, which might fail and the would be difficult to debug.

```
contract TimelockStateRegistry is BaseStateRegistry, ITimelockStateRegistry, ReentrancyGuard {
    function finalizePayload(uint256 timeLockPayloadId_, ...) ... {
        TimelockPayload memory p = timelockPayload[timeLockPayloadId_];
        IBridgeValidator bridgeValidator = IBridgeValidator(superRegistry.getBridgeValidator(p.data.
         ...
        if (p.status != TwoStepsStatus.PENDING) {
            revert Error.INVALID_PAYLOAD_STATUS();
        }
       ...
   }
}
```

Recommendation: Consider checking `timelockPayload[timeLockPayloadId_]` is valid as the first thing of the function.

```
function finalizePayload(uint256 timeLockPayloadId_, ...) ... {
    TimelockPayload memory p = timelockPayload[timeLockPayloadId_];
+   if (p.status != TwoStepsStatus.PENDING) { revert Error.INVALID_PAYLOAD_STATUS(); }
    IBridgeValidator bridgeValidator = IBridgeValidator(superRegistry.getBridgeValidator(p.data.liqD
    ...
-   if (p.status != TwoStepsStatus.PENDING) { revert Error.INVALID_PAYLOAD_STATUS(); }
   ...
 }
```

Superform: Solved in PR 334.
Reviewer: Verified

#I-28 : Terms `two step` and `timelock` both used

Context: TimelockStateRegistry.sol, ERC4626TimelockForm.sol
Description: The following names are used to indicate the two step approach for timelocked vaults, which can be confusing:

- TimelockStateRegistry
- TwoStepsFormRegistry
- TwostepsFormStateRegistry
- onlyTwoStepStateRegistry
- twoStepRegistry

- FormStateRegistry
- ...TWO_STEP_...
- TIMELOCK_...

Recommendation: Consider using a consistent term everywhere.
Superform: Solved in PR 311.
Reviewer: Verified

#I-29 : Code duplication in `dispatchPayload()` and `_dispatchAcknowledgement()`

Context: BaseStateRegistry.sol#L68-L87, CoreStateRegistry.sol#L947-L958, TimelockStateRegistry.sol#L296-L306
Description: The core code of `dispatchPayload()` and `_dispatchAcknowledgement()` is the same. For easier maintenance this could be seperated out in an internal function. Both `CoreStateRegistry` and `TimelockStateRegistry` have identical versions of `_dispatchAcknowledgement()`.

```
abstract contract BaseStateRegistry is IBaseStateRegistry {
    function dispatchPayload(...) ... {
        AMBExtraData memory d = abi.decode(extraData_, (AMBExtraData));
       _dispatchPayload(srcSender_, ambIds_[0], dstChainId_, d.gasPerAMB[0], message_, d.extraDataPe
        if (ambIds_.length > 1) {
            _dispatchProof(srcSender_, ambIds_, dstChainId_, d.gasPerAMB, message_, d.extraDataPerAM
        }
    }
}
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _dispatchAcknowledgement(uint64 dstChainId_, uint8[] memory ambIds_, bytes memory messa
        ...
        AMBExtraData memory d = abi.decode(extraData, (AMBExtraData));
        _dispatchPayload(msg.sender, ambIds_[0], dstChainId_, d.gasPerAMB[0], message_, d.extraDataP
        if (ambIds_.length > 1) {
            _dispatchProof(msg.sender, ambIds_, dstChainId_, d.gasPerAMB, message_, d.extraDataPerAM
        }
    }
 }
contract TimelockStateRegistry is BaseStateRegistry, ITimelockStateRegistry, ReentrancyGuard {
    function _dispatchAcknowledgement(uint64 dstChainId_, uint8[] memory ambIds_, bytes memory messa
        ...
        AMBExtraData memory d = abi.decode(extraData, (AMBExtraData));
        _dispatchPayload(msg.sender, ambIds_[0], dstChainId_, d.gasPerAMB[0], message_, d.extraDataP
        if (ambIds_.length > 1) {
            _dispatchProof(msg.sender, ambIds_, dstChainId_, d.gasPerAMB, message_, d.extraDataPerAM
        }
    }
}
```

Recommendation: Consider moving `_dispatchAcknowledgement()` to `BaseStateRegistry`. Consider moving the core code of `dispatchPayload()` and `_dispatchAcknowledgement()` to an internal function.
Superform: Solved in several PRs.
Reviewer: Verified

#I-30 : Check in `_updateWithdrawPayload()` not obvious

Context: CoreStateRegistry.sol#L551-L645, Abstract.Deploy.s.sol#L144-L146, Abstract.Deploy.s.sol#L383-L390
Description: The check on `getStateRegistryId()` in `_updateWithdrawPayload()` is done for the following reason:

- For deposit payloads, the update always happens on `CoreStateRegistry`
- For withdraw payloads, the update happens on:
    - `CoreStateRegistry` (registryId 1) for forms: 1 ERC4626Form and 3 KYCDaoForm
    - `TimelockStateRegistry` (registryId 2) for forms: 2 ERC4626TimelockForm

So only for `_updateWithdrawPayload()` it is relevant to check the `registryId`. As this is not obvious, it good to add a comment in the source.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _updateWithdrawPayload(...) ... {
        ...
        if (IBaseForm(superform).getStateRegistryId() == _getStateRegistryId(address(this))) {
            ...
        }
    }
}
```

```
abstract contract AbstractDeploy is Script {
    ...
    /// @dev 1 = ERC4626Form, 2 = ERC4626TimelockForm, 3 = KYCDaoForm
    uint32[] public FORM_IMPLEMENTATION_IDS = [uint32(1), uint32(2), uint32(3)];
    string[] public VAULT_KINDS = ["Vault", "TimelockedVault", "KYCDaoVault"];

   registryAddresses[0] = vars.coreStateRegistry;
        registryAddresses[1] = vars.twoStepsFormStateRegistry;
        registryAddresses[2] = vars.broadcastRegistry;

        uint8[] memory registryIds = new uint8[](3);
        registryIds[0] = 1;
        registryIds[1] = 2;
        registryIds[2] = 3;
```

Recommendation: Consider adding a comment in the source explaining the check.
Superform: Solved in PR 324.
Reviewer: Verified

#I-31 : Indents can be reduced

Context: CoreStateRegistry.sol#L598-L603
Description: Some parts of the code have a high nesting level, which makes the code more difficult to read. There are straightforward ways to reduce the indents. Here are some examples where this can be applied.

```
for (uint i; i < len;) {
    if (requirement) {
        ... // actions
}
```

Recommendation: Consider changing the code of the links from the `Context` section above.

```
for (uint i; i < len;) {
-   if (requirement) {
+   if (!requirement) {
+       unchecked { ++i; }
+       continue;
+   }
    ... // actions
-}
```

Superform: Solved by changing the code.
Reviewer: Verified

#I-32 : Incorrect comment in `_updateSingleVaultDepositPayload`

Context: CoreStateRegistry.sol#L512-L548
Description: In function `_updateSingleVaultDepositPayload()`, the first comment about `sets amount to zero` isn't correct as the `amount` is set to `finalAmount_`. The comment seems to be a copy-paste from the `else` clause.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _updateSingleVaultDepositPayload(...) ... {
```

```
        ...
        if (PayloadUpdaterLib.validateSlippage(finalAmount_, singleVaultData.amount, singleVaultDat
            /// @dev sets amount to zero and will mark the payload as UPDATED
            singleVaultData.amount = finalAmount_;
            ...
        } else {
            /// @dev sets amount to zero and will mark the payload as PROCESSED
            singleVaultData.amount = 0;
            ...
        }
    }
}
```

Recommendation: Doublecheck the comment.
Superform: Solved in PR 237.
Reviewer: Verified

#I-33 : Comments in `processPayload()` not accurate

Context: CoreStateRegistry.sol#L188-L271, BroadcastRegistry.sol#L136-L157
Description: The comments in `processPayload()` are not completely accurate.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function processPayload(uint256 payloadId_)
       ...
        v._proof = _message.computeProof();
        /// @dev The number of valid proofs (quorum) must be equal to the required messaging quorum
        if (messageQuorum[v._proof] < _getRequiredMessagingQuorum(v.srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
   }
}
contract BroadcastRegistry is IBroadcastRegistry, QuorumManager {
    function processPayload(uint256 payloadId) external override onlyProcessor {
       ...
        /// @dev The number of valid proofs (quorum) must be equal to the required messaging quorum
        if (messageQuorum[payload_.computeProof()] < getRequiredMessagingQuorum(srcChainId[payloadId
            revert Error.QUORUM_NOT_REACHED();
        }
        ...
    }
}
```

Recommendation: Consider changing the comments to:

```
-/// @dev The number of valid proofs (quorum) must be equal to the required messaging quorum
+/// @dev The number of valid proofs (quorum) must be equal or larger to the required messaging quor
```

Superform: Solved in PR 324.
Reviewer: Verified

#I-34 : `receiveMessage()` uses a hardcoded value

Context: WormholeSRImplementation.sol#L105-L133, Abstract.Deploy.s.sol#L382-L390
Description: The function `receiveMessage()` uses a hardcoded value of `3` to retieve the registry via `getStateRegistry()`. This is inflexible and not immediately clear to the reader of the code.

```
contract WormholeSRImplementation is IBroadcastAmbImplementation {
    function receiveMessage(bytes memory encodedMessage_) public {
        ...
        IBroadcastRegistry targetRegistry = IBroadcastRegistry(superRegistry.getStateRegistry(3));
        ...
    }
}
```

```
abstract contract AbstractDeploy is Script {
    ...
    registryAddresses[0] = vars.coreStateRegistry;
    registryAddresses[1] = vars.twoStepsFormStateRegistry;
    registryAddresses[2] = vars.broadcastRegistry;
    ...
    registryIds[0] = 1;
    registryIds[1] = 2;
    registryIds[2] = 3;
```

Recommendation: Consider using a contant for the value of 3, or supply it via a `constructor`.
Superform: Solved in PR 323.
Reviewer: Verified

#I-35 : Functions `dispatchPayload()` and `broadcastPayload()` use a different pattern

Context: BaseStateRegistry.sol#L141-L201, BaseStateRegistry.sol#L68-L87, BroadcastRegistry.sol#L98-L117, BroadcastRegistry.sol#L184-L220
Description: Function `dispatchPayload()` lets `_dispatchProof()` perform `computeProofBytes()`, while the similar function `broadcastPayload()` performs `computeProofBytes()` itself. This is a slighly different pattern that might be confusing for maintainers and reviewers of the code.

```
abstract contract BaseStateRegistry is IBaseStateRegistry {
    function dispatchPayload(...) ... {
        ....
        _dispatchProof(srcSender_, ambIds_, dstChainId_, d.gasPerAMB, message_, d.extraDataPerAMB);
        ....
    }
    function _dispatchProof(...) ... {
        ....
        data.params = message_.computeProofBytes();
        ....
        tempImpl.dispatchPayload{.... }(srcSender_, dstChainId_, abi.encode(data), overrideData_[i])
        ...
        }
    }
}
contract BroadcastRegistry is IBroadcastRegistry, QuorumManager {
    function broadcastPayload(...) ... {
        ...
        bytes memory proof = message_.computeProofBytes();
        _broadcastProof(srcSender_, ambIds_, d.gasPerAMB, proof, d.extraDataPerAMB);
        ....
    }
    function _broadcastProof(...) ... {
        ...
        tempImpl.broadcastPayload{ value: gasToPay_[i] }(srcSender_, message_, extraData_[i]);
        ...
     }
}
```

Recommendation: Consider using the same pattern.
Superform: Solved in PR 316, changing the broadcast logic.
Reviewer: Verified

#I-36 : `_dispatchPayload()` and `_dispatchProof()` contain duplicate code

Context: BaseStateRegistry.sol#L141-L201
Description: Function `_dispatchProof()` sends a proof via `dispatchPayload()`. The logic to do this is very similar to the function `_dispatchPayload()`. Calling `_dispatchPayload()` from `_dispatchProof()` reduces code duplication and makes maintenance easier.

```
abstract contract BaseStateRegistry is IBaseStateRegistry {
    function _dispatchPayload(...) ... {
```

```
        IAmbImplementation ambImplementation = IAmbImplementation(_getAmbAddress(ambId_));
        if (address(ambImplementation) == address(0)) {
            revert Error.INVALID_BRIDGE_ID();
        }
        ambImplementation.dispatchPayload{ value: gasToPay_ }(srcSender_, dstChainId_, message_, ove
    }
    function _dispatchProof(...) ... {
        ...
        for (uint8 i = 1; i < len;) {
            ...
            IAmbImplementation tempImpl = IAmbImplementation(_getAmbAddress(tempAmbId));
            if (address(tempImpl) == address(0)) {
                revert Error.INVALID_BRIDGE_ID();
            }
            tempImpl.dispatchPayload{ value: gasToPay_[i] }(srcSender_, dstChainId_, abi.encode(data
            ...
        }
    }
}
```

Recommendation: Consider calling `_dispatchPayload()` from `_dispatchProof()`.
Superform: Solved in several PRs including PR 316.
Reviewer: Verified

#I-37 : `msg.sender` check could be modifier

Context: SuperTransmuter.sol#L298-L309, SuperformFactory.sol#L186-L197, SuperRBAC.sol#L101-L117, WormholeSRImplementation.sol#L105-L133, BroadcastRegistry.sol#L120-L133, BaseStateRegistry.sol#L90-L116, WormholeSRImplementation.sol#L77-L103, WormholeARImplementation.sol#L67-L127, HyperlaneImplementation.sol#L77-L98, HyperlaneImplementation.sol#L145-L174, LayerzeroImplementation.sol#L71-L86, LayerzeroImplementation.sol#L134-L164
Description: Several functions have an autorization check on `msg.sender`. On most other locations in the source, such functionality is implemented via modifiers, which could improve readability. Note: Modifiers are inlined so shouldn't increase the gas. Here are the occurances we have found:

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function stateSyncBroadcast(bytes memory data_) external payable override {
        if (msg.sender != superRegistry.getAddress(keccak256("BROADCAST_REGISTRY"))) {
            revert Error.NOT_BROADCAST_REGISTRY();
        }
        ...
    }
}
contract SuperformFactory is ISuperformFactory {
     function stateSyncBroadcast(bytes memory data_) external payable override {
        if (msg.sender != superRegistry.getAddress(keccak256("BROADCAST_REGISTRY"))) {
            revert Error.NOT_BROADCAST_REGISTRY();
        }
        ...
    }
}
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
     function stateSyncBroadcast(bytes memory data_) external override {
        if (msg.sender != superRegistry.getAddress(keccak256("BROADCAST_REGISTRY"))) {
            revert Error.NOT_BROADCAST_REGISTRY();
        }
        ...
    }
}
contract BroadcastRegistry is IBroadcastRegistry, QuorumManager {
    function receiveBroadcastPayload(uint64 srcChainId_, bytes memory message_) external override {
        if (!superRegistry.isValidBroadcastAmbImpl(msg.sender)) {
            revert Error.NOT_BROADCAST_AMB_IMPLEMENTATION();
        }
        ...
```

```
    }
}
abstract contract BaseStateRegistry is IBaseStateRegistry {
     function receivePayload(uint64 srcChainId_, bytes memory message_) external override {
        if (!superRegistry.isValidAmbImpl(msg.sender)) {
            revert Error.NOT_AMB_IMPLEMENTATION();
        }
       ...
    }
}

contract WormholeSRImplementation is IBroadcastAmbImplementation {
    function receiveMessage(bytes memory encodedMessage_) public {
        if (
            !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole(
                keccak256("WORMHOLE_VAA_RELAYER_ROLE"), msg.sender
            )
        ) {
            revert Error.CALLER_NOT_RELAYER();
        }
       ...
    }
    function broadcastPayload(...) ... {
        if (!superRegistry.isValidStateRegistry(msg.sender)) {
            revert Error.NOT_STATE_REGISTRY();
        }
       ...
    }
}
contract WormholeARImplementation is IAmbImplementation, IWormholeReceiver {
    function dispatchPayload(...) ... {
        if (!superRegistry.isValidStateRegistry(msg.sender)) {
            revert Error.NOT_STATE_REGISTRY();
        }
        ...
    }
   function receiveWormholeMessages(...) ... {
        if (msg.sender != address(relayer)) {
            revert Error.CALLER_NOT_RELAYER();
        }
        ...
    }
}
 contract HyperlaneImplementation is IAmbImplementation, IMessageRecipient {
    function dispatchPayload(...) ... {
        if (!superRegistry.isValidStateRegistry(msg.sender)) {
            revert Error.NOT_STATE_REGISTRY();
        }
        ...
    }
    function handle(uint32 origin_, bytes32 sender_, bytes calldata body_) external override {
        if (msg.sender != address(mailbox)) {
            revert Error.CALLER_NOT_MAILBOX();
        }
        ...
    }
}
contract LayerzeroImplementation is IAmbImplementation, ILayerZeroUserApplicationConfig, ILayerZeroR
     function dispatchPayload(...) ... {
        if (!superRegistry.isValidStateRegistry(msg.sender)) {
            revert Error.NOT_STATE_REGISTRY();
        }
        ...
    }
    function lzReceive(...) ... {
        if (msg.sender != address(lzEndpoint)) {
            revert Error.CALLER_NOT_ENDPOINT();
        }
        ...
```

```
    }
}
```

Recommendation: Consider using a modifier for the checks on `msg.sender`.
Superform: Solved in PR 330.
Reviewer: Verified

#I-38 : `stateMultiSync()` and `stateSync()` in `SuperPositions` and `SuperTransmuter` very similar

Context: SuperPositions.sol#L128-L219, SuperTransmuter.sol#L197-L295
Description: The versions of `stateMultiSync()` and `stateSync()` in `SuperPositions` and `SuperTransmuter` are very similar except for the minting part. As the similar parts are relatively complicated it might be good to combine them for easier maintenance.

```
contract SuperPositions is ISuperPositions, ERC1155A, StateSyncer {
    function stateMultiSync(AMBMessage memory data_) ... {
       ...
       _batchMint(srcSender, returnData.superformIds, returnData.amounts, "");
       ...
    }
    function stateSync(AMBMessage memory data_) ... {
       ...
       _mint(srcSender, returnData.superformId, returnData.amount, "");
       ...
    }
}
```

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function stateMultiSync(AMBMessage memory data_) ... {
        ...
        uint256 len = returnData.superformIds.length;
        for (uint256 i; i < len;) {
            sERC20(synthethicTokenId[returnData.superformIds[i]]).mint(srcSender, returnData.amounts
            ...
        }
    ...
    }
    function stateSync(AMBMessage memory data_) ... {
        ...
        sERC20(synthethicTokenId[returnData.superformId]).mint(srcSender, returnData.amount);
        ...
     }
}
```

Recommendation: Consider combining the versions `stateMultiSync()` and `stateSync()` and move them to `StateSyncer`. Move the minting logic to a seperate function that is different for `SuperPositions` and `SuperTransmuter`.
Superform: Solved by combining `SuperPositions` and `SuperTransmuter` in several PRs.
Reviewer: Verified

#I-39 : No emit in `registerTransmuter()` and `_deployTransmuter()`

Context: SuperTransmuter.sol#L92-L125, SuperTransmuter.sol#L329-L342
Description: The function `registerTransmuter()` and `_deployTransmuter()` don't `emit` an event on the deployment of and `sERC20`. This might be useful for offchain indexing.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(uint256 superformId_, bytes memory extraData_) external override ret
        ...
        synthethicTokenId[superformId_] = address(new sERC20(...));
        ...
    }
    function _deployTransmuter(bytes memory message_) internal {
```

```
        ...
        address syntheticToken = address(new sERC20(...));
        ...
    }
```

Recommendation: Consider emitting an event in `registerTransmuter()` and `_deployTransmuter()`.
Superform: Solved in PR 330.
Reviewer: Verified

#I-40 : Broadcast messages to new chains

Context: SuperTransmuter.sol#L318-L326, SuperformFactory.sol#L279-L287, SuperRBAC.sol#L151-L159
Description: If a new chain is added to the protocol, its important that at least some of the broadcasted messages are also processed on that chain to guarantee a consistent state.
According to the project: We use wormhole's specialized relayer where we don't specify the destination chain, hence we could re-use the VAAs again on new chains and could sync the transmutters.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function _broadcast(bytes memory message_, bytes memory extraData_) internal {
        ...
        ...broadcastPayload{...}(...);
    }

contract SuperformFactory is ISuperformFactory {
    function _broadcast(bytes memory message_, bytes memory extraData_) internal {
        ...
        ...broadcastPayload{...}(...);
    }


contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    function _broadcast(bytes memory message_, bytes memory extraData_) internal {
        ...
        ...broadcastPayload{...}(...);
    }
```

Recommendation: Implement a way to select and transmit broadcast messages to new chains, using VAAs.
Superform: On the new chain, we just call the `receiveMessage()` function on the `SR implementation`. We don't need to re-initiate it from the source chain. So the new chain `WormholeSRImplementation` validates the emitter address and sync the state there.
Reviewer: Acknowledged

#I-41 : Comments about `multi` confusing

Context: SuperPositions.sol#L128-L219, SuperTransmuter.sol#L197-L295
Description: The comments about `multi` in the functions `stateMultiSync()` and `stateSync()` of `SuperPositions` and `SuperTransmuter` are slightly confusing.

```
function stateMultiSync(AMBMessage memory data_)
    ...
    /// @dev verify this is a single vault mint
    if (multi == 0) revert Error.INVALID_PAYLOAD();
    ...
}
function stateSync(AMBMessage memory data_)
    ...
    /// @dev verify this is a multi vault mint
    if (multi == 1) revert Error.INVALID_PAYLOAD();
    ...
}
```

Recommendation: Consider changing the comments to:

```
function stateMultiSync(AMBMessage memory data_)
    ...
-   /// @dev verify this is a single vault mint
+   /// @dev verify this is a not single vault mint
    if (multi == 0) revert Error.INVALID_PAYLOAD();
    ...
}
function stateSync(AMBMessage memory data_)
    ...
-   /// @dev verify this is a multi vault mint
+   /// @dev verify this is a not multi vault mint
    if (multi == 1) revert Error.INVALID_PAYLOAD();
    ...
}
```

Superform: Solved in PR 330.
Reviewer: Verified

#I-42 : `_kycCheck()` could be modifier

Context: ERC4626KYCDaoForm.sol#L13-L92
Description: In contract `ERC4626KYCDaoForm`, the function `_kycCheck()` is used to check authorization. On most other locations in the source, similar functionality is implemented via modifiers.

```
contract ERC4626KYCDaoForm is ERC4626FormImplementation {
  function _kycCheck(address srcSender_) internal view {
        if (!kycDAO4626(vault).kycCheck(srcSender_)) revert NO_VALID_KYC_TOKEN();
    }

    function _directDepositIntoVault(...) ... {
        _kycCheck(srcSender_);
       ...
    }
    function _directWithdrawFromVault(...) ... {
        _kycCheck(srcSender_);
       ...
    }
    function _xChainDepositIntoVault(...) ... {
        _kycCheck(srcSender_);
       ...
    }
    function _xChainWithdrawFromVault(...) ... {
        _kycCheck(srcSender_);
       ...
    }
```

Recommendation: Consider making a modifier for `_kycCheck()`.
Superform: Solved in PR 311.
Reviewer: Verified

#I-43 : Parameters for constructor of `ERC4626FormImplementation` not descriptive

Context: ERC4626Form.sol#L10-L14, ERC4626KYCDaoForm.sol#L13-L25, ERC4626TimelockForm.sol#L18-L35
Description: The contructor of `ERC4626FormImplementation` is called with the values `1` and `2`. It would be clearer to use constants for these values.

```
contract ERC4626Form is ERC4626FormImplementation {
    constructor(address superRegistry_) ERC4626FormImplementation(superRegistry_, 1) { }
    ...
}
contract ERC4626KYCDaoForm is ERC4626FormImplementation {
    ...
    constructor(address superRegistry_) ERC4626FormImplementation(superRegistry_, 1) { }
    ...
```

```
}
contract ERC4626TimelockForm is ERC4626FormImplementation {
    ...
    constructor(address superRegistry_) ERC4626FormImplementation(superRegistry_, 2) { }
    ...
}
```

Recommendation: Use constants for the parameters to the constructor for `ERC4626FormImplementation`.
Superform: Solved in PR 311.
Reviewer: Verified

#I-44 : Reliance on `UPDATER_KEEPER`

Context: CoreStateRegistry.sol#L713-L780, CoreStateRegistry.sol#L837-L884
Description: The liquidity bridge (which is currently based on LI.FI) doesn't transmit the type of token and the amount of tokens. Due to this the functions `_processSingleDeposit()` and `_processMultiDeposit()` can't check the type of tokens that are transferred. This gap is filled by the `UPDATER_KEEPER` that validates and updates the amount, then there is the `PROCESSOR_KEEPER` that calls the `_processSingleDeposit()`. This introduces off chain dependencies and thus centralization risk.
The goal of the protocol is however to further increase decentralization in the future, while also reducing dependencies on individual bridges.
Recommendation: In order to increase decentralization and reducing dependencies on individual bridges and offchain components we see the following potential solutions which might be introduced in future versions of the protocol:

- use bridges that have `xchain` callbacks, like xchain-zaps from LI.FI. However this limits the number of bridges that can be used;
- use a small number of tokens that are bridged, for example just USDC. Then the tokens can be whitelisted and only the amount of tokens has to be transported, which can be done via de AMB. However the slippage costs will be higher;
- use a `bucket` mechanism, where each liquitidity transfer is done to a seperate `bucket`, this prevents accidentally mixings funds of users. The `bucket` can be recycled when the xchain processing is finished, to limit the number of `buckets` in use.

Note: After discussing with the project: the core issue with checking tokens across chains is that the token address are different on each chain.
Superform: This can't be fixed with the current design, we will think of this redesign for a future version of the protocol.
Reviewer: Acknowledged

#I-45 : Incorrect/unexpected messages are ignored

Context: SuperformFactory.sol#L186-L197, CoreStateRegistry.sol#L188-L271, SuperRBAC.sol#L101-L117, SuperTransmuter.sol#L197-L309, SuperPositions.sol#L128-L219
Description: The functions that receive crosschain messages like `processPayload()`, `stateSync()`, `stateMultiSync()` and `stateSyncBroadcast()` trigger on expected and correct messages. If an incorrect/unexpected message is received it is ignored. For defensive code it might be good to revert on incorrect/unexpected messages.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function processPayload(uint256 payloadId_)
        ...
        if (v.callbackType == uint256(CallbackType.RETURN) || v.callbackType == uint256(CallbackType
              ...
        }
        if (v.callbackType == uint8(CallbackType.INIT)) {
            ...
        }
    }
}
```

Recommendation: Consider to revert in the functions `processPayload()`, `stateSync()`, `stateMultiSync()` and `stateSyncBroadcast()` when an incorrect/unexpected message is received.
Superform: Solved in PR 332, note `stateSyncBroadcast` don't have any callbacktype.
Reviewer: Verified

#I-46 : Shadowing variables `v.len` and `len`

Context: BaseRouterImplementation.sol#L922-L1004
Description: Function `_multiVaultTokenForward()` uses both `v.len` and `len`, which could be confusing.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _multiVaultTokenForward(...) ... {
        ...
        for (uint256 i; i < v.len;) {
            ...
            uint256 len = vaultData_.liqData[i].txData.length;
            if (len == 0) {
               ...
            } else {
              ...
            }
      }
}
```

Recommendation: As `len` is only used once, it can be removed in the following way:

```
-   uint256 len = vaultData_.liqData[i].txData.length;
-   if (len == 0) {
+   if (vaultData_.liqData[i].txData.length== 0) {
    ...
}
```

Superform: Solved in PR 338.
Reviewer: Verified

#I-47 : Inaccurate parameter name in function `_singleVaultTokenForward()`

Context: BaseRouterImplementation.sol#L855-L1004
Description: The function `_singleVaultTokenForward()` can also sets an allowance for bridges so the `superform_` isn't accurate and could be confusing. The comparable function `_multiVaultTokenForward()` uses `targets_` which is easier to understand.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _singleVaultTokenForward(address srcSender_,address superform_, ...) ... {
        ...
    }
    function _multiVaultTokenForward(address srcSender_,address[] memory targets_, ...) ...{
        ...
     }
}
```

Recommendation: Consider changing the code in the following way:

```
-   function _singleVaultTokenForward(address srcSender_,address superform_, ...) ... {
+   function _singleVaultTokenForward(address srcSender_,address target_, ...) ... {
        ...
    }
```

Superform: Solved in PR 282.
Reviewer: Verified

#I-48 : Retrieved addresses not checked for 0

Context: SuperRegistry.sol#L238-L240

Description: Frequently an address is retrieved and then the next function is called directly. If the address isn't found and thus an `address(0)` is return, then the next call will revert without a clear message. This make debugging more difficult.

```
StateSyncer.sol:              if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
StateSyncer.sol:              !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
SuperformFactory.sol:         if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
SuperformFactory.sol:         if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
SuperformFactory.sol:         IBroadcastRegistry(superRegistry.getAddress(keccak256("BROADCAST_REGIS
SuperPositions.sol:           !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
SuperPositions.sol:           !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
SuperTransmuter.sol:          !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
SuperTransmuter.sol:          !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
SuperTransmuter.sol:          IBroadcastRegistry(superRegistry.getAddress(keccak256("BROADCAST_REGIS
SuperRBAC.sol:                IBroadcastRegistry(superRegistry.getAddress(keccak256("BROADCAST_REGIS
PayMaster.sol:                !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
PaymentHelper.sol:            if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
PaymentHelper.sol:            if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
PaymentHelper.sol:            nextPayloadId = ReadOnlyBaseRegistry(superRegistry.getAddress(keccak25
LiFiValidator.sol:            if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
DstSwapper.sol:               !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
DstSwapper.sol:               if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
BaseRouterImplementation.sol: IBaseStateRegistry(superRegistry.getAddress(keccak256("CORE_STATE_REGI
BaseRouterImplementation.sol: ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTOR
BaseRouterImplementation.sol: ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTOR
BaseRouterImplementation.sol: ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTOR
BaseRouterImplementation.sol: ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTOR
BaseRouterImplementation.sol: ISuperformFactory(superRegistry.getAddress(keccak256("SUPERFORM_FACTOR
BaseRouterImplementation.sol: IPayMaster(superRegistry.getAddress(keccak256("PAYMASTER"))).makePayme
BaseForm.sol:                 if (IFormBeacon(ISuperformFactory(superRegistry.getAddress(keccak256("
TimelockStateRegistry.sol:    !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
BroadcastRegistry.sol:        !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
BroadcastRegistry.sol:        if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
BroadcastRegistry.sol:        !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
BroadcastRegistry.sol:        Target(superRegistry.getAddress(targetId)).stateSyncBroadcast(payload_
LayerzeroImplementation.sol:  if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
WormholeARImplementation.sol: if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
WormholeSRImplementation.sol: !ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).hasRole
HyperlaneImplementation.sol:  if (!ISuperRBAC(superRegistry.getAddress(keccak256("SUPER_RBAC"))).has
```

```
contract SuperRegistry is ISuperRegistry, QuorumManager {
    function getAddress(bytes32 id_) external view override returns (address) {
        return registry[id_][uint64(block.chainid)];
    }
```

```
BaseForm.sol:  return IFormBeacon(ISuperformFactory(...).getFormBeacon(formBeaconId_)).paused() == 1
```

Recommendation: Consider doing one of the following:

- always check the result of functions like:
    - `getAddress()`
    - `getFormBeacon()`
    - `getBridgeValidator()`
    - `getBridgeAddress()`
    - `getAmbAddress()`
    - `getStateRegistry()`
    - `getStateSyncer()`
    - `PERMIT2()`
- inside these functions revert if the address isn't found.
- have a variation these functions that reverts if the address isn't found.
- although currently not used in a risky way, for consistency also include these functions:
- `getAddressByChainId()`
- `getRouter()`

Superform: Solved in PR 331.

Reviewer: Verified

#I-49 : Paused values allow for mistakes

Context: BaseRouterImplementation.sol#L712-L834, FormBeacon.sol#L24, SuperformFactory.sol#L209-L211

Description: The values for `paused` are not documented and are not obvious. This increases the probability of mistakes.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _validateSuperformData(...) ... {
        ...
        return IFormBeacon(...).getFormBeacon(...)).paused() == 1; // what is the meaning of 1?
    }
    function _validateSuperformsDepositData(...) ... {
        ...
        if (IFormBeacon(...).getFormBeacon(...)).paused() == 2) return false; // what is the meaning
        ...
    }
    function _validateSuperformsWithdrawData(...) ... {
        ...
        if (IFormBeacon(...).getFormBeacon(...)).paused() == 2) return false; // what is the meaning
        ...
    }
}

contract FormBeacon is IFormBeacon {
    uint256 public paused = 1;
}

contract SuperformFactory is ISuperformFactory {
    function isFormBeaconPaused(uint32 formBeaconId_) external view override returns (uint256 paused
        paused_ = FormBeacon(formBeacon[formBeaconId_]).paused();
    }
```

Recommendation: Consider changing the code to:

```
contract FormBeacon is IFormBeacon {
-   uint256 public paused = 1;
+   uint256 public paused = 1;   // 1: not paused, 2: paused
+   function isPaused() external view returns(bool) {
+       return paused == 2;
+   }
}

abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _validateSuperformData(...) ... {
        ...
-       return IFormBeacon(...).getFormBeacon(...)).paused() == 1;
+       return !IFormBeacon(...).getFormBeacon(...)).isPaused();  // note the !
    }
    function _validateSuperformsDepositData(...) ... {
        ...
-       if (IFormBeacon(...).getFormBeacon(...)).paused() == 2) return false;
+       if (IFormBeacon(...).getFormBeacon(...)).isPaused()) return false;
       ...
    }
}

contract SuperformFactory is ISuperformFactory {
-   function isFormBeaconPaused(uint32 formBeaconId_) external view override returns (uint256 paused
+   function isFormBeaconPaused(uint32 formBeaconId_) external view override returns (bool paused_)
-       paused_ = FormBeacon(formBeacon[formBeaconId_]).paused();
+       paused_ = FormBeacon(formBeacon[formBeaconId_]).isPaused();
    }
}
```

Note: `isFormBeaconPaused()` can also be used after its updated. Note: same for `_validateSuperformsWithdrawData()`.
Superform: Solved in PR 227.
Reviewer: Verified

#I-50 : Typos

Context: BaseRouterImplementation.sol#L712-L733, ERC4626FormImplementation.sol#L175, CoreStateRegistry.sol#L312-L336, DataTypes.sol#L94-L104, PaymentHelper.sol#L81
Description: There are a few typos in the code and comments:

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _validateSuperformData(...) ... {
        ...
        /// destinatiom) /// typo ===> destination (m->n)
        ...
    }
}
abstract contract ERC4626FormImplementation is BaseForm, LiquidityHandler {
    struct ProcessDirectWithdawLocalVars { /// typo ===> ProcessDirectWithdrawLocalVars (daw->draw)
        ...
    }
}
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function disputeRescueFailedDeposits(uint256 payloadId_) external override {
        ...
        revert Error.INVALID_DISUPTER();   /// typo ===> INVALID_DISPUTER (UP->PU)
        ...
    }
}
```

DataTypes.sol:

```
struct InitMultiVaultData {
    ...
    uint256[] superformIds;
    uint256[] amounts;
    uint256[] maxSlippage;  /// typo ===> maxSlippages  (same as in MultiVaultSFData)
    ...
}

interface ReadOnlyBaseRegistry is IBaseStateRegistry {
    ...
    PREVILAGES ADMIN ONLY FUNCTIONS /// typo ===> PRIVILEGES (E -> I, A->E)
    ...
}
```

Recommendation: Consider fixing the typos listed above.
Superform: Solved in PR 330.
Reviewer: Verified

#I-51 : Functions `_validateSuperformData()` , `_validateSuperformsDepositData()` and `_validateSuperformsWithdrawData()` can be optimized

Context: BaseRouterImplementation.sol#L712-L834
Description: The functions `_validateSuperformData()` , `_validateSuperformsDepositData()` and `_validateSuperformsWithdrawData()` can be combined and optimized. The mains goals for this are increased maintainability and readability and possibly reduction of gas.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
  function _validateSuperformData(...) ... {
        if (dstChainId_ != DataLib.getDestinationChain(superformData_.superformId)) return false;
        if (superformData_.maxSlippage > 10_000) return false;
        (, uint32 formBeaconId_,) = superformData_.superformId.getSuperform();
```

```
        return IFormBeacon(ISuperformFactory(...).getFormBeacon(...)).paused() == 1;
    }
    function _validateSuperformsDepositData(...) ... {
        ...
        for (uint256 i; i < len;) {
            /// @dev 10000 = 100% slippage
            if (superformsData_.maxSlippages[i] > 10_000) return false;
            (, uint32 formBeaconId_, uint64 sfDstChainId) = superformsData_.superformIds[i].getSuper
            if (dstChainId_ != sfDstChainId) return false;
            if (IFormBeacon(ISuperformFactory(...)).getFormBeacon(...).paused() == 2 ) return false;
            ...
    }
    function _validateSuperformsWithdrawData(...) ... {
        ... // same as _validateSuperformsDepositData
    }
}
```

Recommendation: Here are some optimization suggestions:

1. retrieve `formBeaconId_` and `sfDstChainId` in one go, like `_validateSuperformsDepositData()` does;
2. combine `_validateSuperformsDepositData()` and `_validateSuperformsWithdrawData()` as they are the same;
3. replace the inside of the `for` loop of `_validateSuperformsDepositData()` with a call to `_validateSuperformData()` to make sure the checks are the same. Note this will take some extra gas;
4. retrieve `SuperformFactory` outside of the for loop;
5. first collect all the different `formBeacons`, of which there currently at most 3. Only the call the `paused()` function for each different `formBeacon`.

Superform: Solved in PR 329.
Reviewer: Verified

#I-52 : Inconsistent placement of `burn` functions

Context: BaseRouterImplementation.sol#L228-L348, BaseRouterImplementation.sol#L381-L412
Description: The functions `_singleXChainMultiVaultWithdraw()` and `_singleDirectMultiVaultWithdraw()` directly burn tokens, however the comparable functions `_singleXChainSingleVaultWithdraw()` and `_singleDirectSingleVaultWithdraw()` do this via function `_buildWithdrawAmbData()`. This is inconsistent and the function name of `_buildWithdrawAmbData()` doesn't suggest the burning.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _singleXChainMultiVaultWithdraw(SingleXChainMultiVaultStateReq memory req_) internal vi
        ...
        IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnBatch(...);
        ...
    }
    function _singleXChainSingleVaultWithdraw(SingleXChainSingleVaultStateReq memory req_) internal
        ...
        (ambData, vars.currentPayloadId) = _buildWithdrawAmbData(...);
        ...
    }
    function _singleDirectSingleVaultWithdraw(SingleDirectSingleVaultStateReq memory req_) internal
        ...
        (ambData, vars.currentPayloadId) = _buildWithdrawAmbData(...);
        ...
    }
    function _singleDirectMultiVaultWithdraw(SingleDirectMultiVaultStateReq memory req_) internal vi
        ...
        IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnBatch(...);
        ...
    }
    function _buildWithdrawAmbData(...) ... {
        ...
        IStateSyncer(superRegistry.getStateSyncer(ROUTER_TYPE)).burnSingle(...);
        ...
```

```
    }
}
```

Recommendation: Move the `burnSingle()` statement from `_buildWithdrawAmbData()` to `_singleXChainSingleVaultWithdraw()` and `_singleDirectSingleVaultWithdraw()`. Also see issue Local deposits and withdraws don't check the paused state of forms .
Superform: Solved by PR 298.
Reviewer: Verified

Severity: Gas Optimization

#G-1 : Functions `_getGasPrice()` and `_getNativeTokenPrice()` can be optimized

Context: PaymentHelper.sol#L740-L762
Description: The functions `_getGasPrice()` and `_getNativeTokenPrice()` retrieve the oracle address twice. This could be optimized.

```
contract PaymentHelper is IPaymentHelper {
    function _getGasPrice(uint64 chainId_) internal view returns (uint256) {
        if (address(gasPriceOracle[chainId_]) != address(0)) {
            (, int256 value,, uint256 updatedAt,) = gasPriceOracle[chainId_].latestRoundData();
            ...
         }
         ...
    }
    function _getNativeTokenPrice(uint64 chainId_) internal view returns (uint256) {
         if (address(nativeFeedOracle[chainId_]) != address(0)) {
             (, int256 dstTokenPrice,, uint256 updatedAt,) = nativeFeedOracle[chainId_].latestRoundD
             ...
        }
        ...
    }
}
```

Recommendation: Consider storing the oracle address in a temporary variable.
Superform: Solved in PR 289.
Reviewer: Verified

#G-2 : Not all return data from `_estimateAMBFees()` is used

Context: PaymentHelper.sol#L506-L537
Description: Function `_estimateAMBFees()` also returns `feeSplitUp[]`, however this is never used. Note: The `feeSplitUp[]` from the comparable function `_estimateAMBFeesReturnExtraData()` is used.

```
contract PaymentHelper is IPaymentHelper {
    function _estimateAMBFees(...) ... {
        feeSplitUp = new uint256[](len);
        for (uint256 i; i < len;) {
            ...
            feeSplitUp[i] = tempFee;
            ...
        }
    }
}
```

Recommendation: Consider removing the `feeSplitUp[]` calculation from `_estimateAMBFees()`.
Superform: Solved by PR 256.
Reviewer: Verified

#G-3 : Realistic input `estimateFees()` might not be neccesary

Context: PaymentHelper.sol#L506-L537

Description: Function `_estimateAMBFees()` send realistic data to `estimateFees()`, which use a relative large amount of gas. The implementations, for example layerzero, just seem to use the length. Note: assuming these functions are only called off chain gas efficiency isn't very important.

```
contract PaymentHelper is IPaymentHelper {
    function _estimateAMBFees(...) ... {
        ...
        bytes memory proof_ = abi.encode(AMBMessage(type(uint256).max, abi.encode(keccak256(message_
        ...
        uint256 tempFee = IAmbImplementation(superRegistry.getAmbAddress(ambIds_[i])).estimateFees(
                dstChainId_, i != 0 ? proof_ : message_, extraDataPerAMB[i]);
        ...
    }
}
```

Recommendation: Consider only using the length of the message. The function `generateSingleVaultMessage()` and `generateMultiVaultMessage()` wouldn't be needed then either.
Superform: This *might* change in the future, and we'd rather keep ability to estimate on the entire message (as you noted, these functions will be called off-chain anyway by our backend to display to users)
Reviewer: Acknowledged

#G-4 : Function `_generateExtraData()` can be optimized

Context: PaymentHelper.sol#L469-L503
Description: Function `_generateExtraData()` retrieves `gasPerKB[dstChainId_]` twice. This can be optimized.

```
contract PaymentHelper is IPaymentHelper {
    function _generateExtraData(...) ... {
        ...
        uint256 totalDstGasReqInWei = message_.length * gasPerKB[dstChainId_];
        uint256 totalDstGasReqInWeiForProof = 32 * gasPerKB[dstChainId_];
        ...
    }
}
```

Recommendation: Consider storing the value of `gasPerKB[dstChainId_]` in a temporary variable.
Superform: Solved in PR 307.
Reviewer: Verified

#G-5 : Unnecessary assignment in `estimateSingleDirectSingleVault()` and `estimateSingleDirectMultiVault()`,

Context: PaymentHelper.sol#L380-L432
Description: In the functions `estimateSingleDirectSingleVault()` and `estimateSingleDirectMultiVault()`, `dstAmount` isn't assigned a value so it is already 0. So there is no need to assign it.

```
contract PaymentHelper is IPaymentHelper {
    function estimateSingleDirectSingleVault(...) ... {
        ...
        dstAmount = 0;
    }
    function estimateSingleDirectMultiVault(...) ... {
        ...
        dstAmount = 0;
    }
}
```

Recommendation: Consider removing `dstAmount = 0`.
Superform: Solved in PR 290.
Reviewer: Verified

#G-6 : Function `estimateSingleDirectMultiVault()` can be optimized

Context: PaymentHelper.sol#L380-L432
Description: In function `estimateSingleDirectMultiVault()`, the value for `twoStepCost[uint64(block.chainid)] * _getGasPrice(uint64(block.chainid))` is constant, so it can be taken outside of the for loop to save some gas.

```
contract PaymentHelper is IPaymentHelper {

    function estimateSingleDirectMultiVault(...) ... {
        ...
        for (uint256 i; i < len;) {
             ...
             srcAmount += twoStepCost[uint64(block.chainid)] * _getGasPrice(uint64(block.chainid));
             ...
        }
        ...
    }
}
```

Recommendation: Consider changing the code to:

```
function estimateSingleDirectMultiVault(...) ... {
    ...
+   uint twoStepPrice = twoStepCost[uint64(block.chainid)] * _getGasPrice(uint64(block.chainid));
    for (uint256 i; i < len;) {
        ...
-       srcAmount += twoStepCost[uint64(block.chainid)] * _getGasPrice(uint64(block.chainid));
+       srcAmount += twoStepPrice;
        ...
    }
}
```

Superform: Solved in PR 292.
Reviewer: Verified

#G-7 : Function `estimateMultiDstMultiVault()` could use `len`

Context: PaymentHelper.sol#L196-L247
Description: In function `estimateMultiDstMultiVault()`, the call to `_estimateAckProcessingCost()`, uses `req_.dstChainIds.length`. It could also use `len`.

```
interface ReadOnlyBaseRegistry is IBaseStateRegistry {
    function estimateMultiDstMultiVault(...) ... {
        uint256 len = req_.dstChainIds.length;
        ...
        srcAmount += _estimateAckProcessingCost(req_.dstChainIds.length, superformIdsLen);
        ...
     }
}
```

Recommendation: Consider changing the code to:

```
function estimateMultiDstMultiVault(...) ... {
    uint256 len = req_.dstChainIds.length;
    ...
-     srcAmount += _estimateAckProcessingCost(req_.dstChainIds.length, superformIdsLen);
+     srcAmount += _estimateAckProcessingCost(len, superformIdsLen);
    ...
}
```

Superform: Solved in PR 293.
Reviewer: Verified

#G-8 : An `emit` in function `setPermit2()` can be optimized

Context: File.sol#L123 https://github.com/superform-xyz/superform-core/blob/2fa594b01e6c970200672a9b79018c11084032e6/src/settings/SuperRegistry.sol#L97-L104
Description: Function `setPermit2()` reads a storage variable it has just set. Some gas could be saved here.

```
contract SuperRegistry is ISuperRegistry, QuorumManager {
    address public PERMIT2;
    ...
    function setPermit2(address permit2_) external override onlyProtocolAdmin {
        ...
        PERMIT2 = permit2_;
        emit SetPermit2(PERMIT2);
    }
}
```

Recommendation: Consider changing the code to:

```
function setPermit2(address permit2_) external override onlyProtocolAdmin {
    ...
-   emit SetPermit2(PERMIT2);
+   emit SetPermit2(permit2_);
}
```

Superform: Solved by PR 264.
Reviewer:

#G-9 : `revokeRoleSuperBroadcast()` and `stateSyncBroadcast()` use different revoke functions

Context: SuperRBAC.sol#L78-L117, AccessControl.sol#L160-L162
Description: Function `revokeRoleSuperBroadcast()` uses `revokeRole()`, while its mirror function on the xchains `stateSyncBroadcast()`, uses `_revokeRole()`. `revokeRoleSuperBroadcast()` could also call `_revokeRole()` which saves a small amount of gas and is more consistent.

```
contract SuperRBAC is ISuperRBAC, AccessControlEnumerable {
    function revokeRoleSuperBroadcast(..., address addressToRevoke_, ...) ... onlyRole(PROTOCOL_ADMI
        ...
        revokeRole(role_, addressToRevoke_);
         ...
    }
    function stateSyncBroadcast(bytes memory data_) external override {
        ...
        _revokeRole(role, addressToRevoke);
    }
 }
abstract contract AccessControl is Context, IAccessControl, ERC165 {
    function revokeRole(bytes32 role, address account) public virtual override onlyRole(getRoleAdmir
        _revokeRole(role, account);
    }
```

Recommendation: Consider to let function `revokeRoleSuperBroadcast()` also use `_revokeRole()`.
Superform: Solved in PR 301.
Reviewer: Verified

#G-10 : Function `validateSuperformChainId()` can call `getDestinationChain()`

Context: DataLib.sol#L104-L118, DataLib.sol#L48-L56
Description: Function `validateSuperformChainId()` calls `getSuperform()` and ignores some of the results. It could also call `getDestinationChain()`, which saves some gas.

```
library DataLib {
    function validateSuperformChainId(uint256 superformId_, uint64 chainId_) internal pure {
        (,, uint64 chainId) = getSuperform(superformId_);
        ...
    }
```

```
    function getSuperform(uint256 superformId_) ... returns( ..., uint64 chainId_) {
        ...
        chainId_ = uint64(superformId_ >> 192);
    }
    function getDestinationChain(uint256 superformId_) internal pure returns (uint64 chainId_) {
        chainId_ = uint64(superformId_ >> 192);
    }
}
```

Recommendation: Consider calling `getDestinationChain()`.
Superform: Solved by PR 265.
Reviewer: Verified

#G-11 : Not all results of `getSuperforms()` are used

Context: DataLib.sol#L63-L99
Description: Functions `getSuperforms()` used relatively complicated assembly and is only used retrieve `superforms[]`. It could be simplified to only return that, which will save gas.
Uses of `getSuperforms()`:

```
BaseRouterImplementation.sol:
(v.superforms,,)                = DataLib.getSuperforms(vaultData_.superformIds);
(address[] memory superforms,,) = DataLib.getSuperforms(vaultData_.superformIds);

CoreStateRegistry.sol:
(address[] memory superforms,,) = DataLib.getSuperforms(multiVaultData.superformIds);
```

```
library DataLib {
    function getSuperforms(uint256[] memory superformIds_) ...
        returns (address superform_, uint32 formBeaconId_, uint64 chainId_) {
        ...
        assembly ("memory-safe") {
           ...
        }
    }
}
```

Recommendation: Consider simplifying function `getSuperforms()` or even consider only using calls to `getSuperform()` and remove function `getSuperforms()`.
Superform: Solved in PR 314.
Reviewer: Verified

#G-12 : `nonReentrant` is set and reset in a loop

Context: DstSwapper.sol#L70-L155
Description: Function `batchProcessTx()` repeatedly calls `processTx()` and everytime the `nonReentrant` is set and reset. This is relatively expensive.

```
contract DstSwapper is IDstSwapper, ReentrancyGuard {
     function batchProcessTx(...) external ... {
        ...
        for (uint256 i; i < len;)
            processTx(...);
            ...
        }
    }
    function processTx(...) public ... nonReentrant {
        ...
    }
}
```

Recommendation: Consider using an `internal` function without `nonReentrant` and call this from two `external` functions with `nonReentrant`. Here is an example:

```
contract DstSwapper is IDstSwapper, ReentrancyGuard {
-   function batchProcessTx(...) external ... {
+   function batchProcessTx(...) external ... nonReentrant  {
        ...
        for (uint256 i; i < len;)
-           processTx(...);
+           _processTx(...);
            ...
        }
    }
-    function processTx(...) public ... nonReentrant {
+    function processTx(...) external ... nonReentrant {
-        ... // actions
+       _processTx(...)
    }
+   function _processTx(...) internal ... {
+       ... // actions
+   }
}
```

Superform: Solved in PR 274.
Reviewer: Verified

#G-13 : `Quorum` check done at end of function

Context: TimelockStateRegistry.sol#L186-L218
Description: The function `processPayload()` of `TimelockStateRegistry` does the `Quorum` check as the last thing in the function. If this fails all other actions have to be reverted. The comparable function `processPayload()` of `CoreStateRegistry` does this earlier in the function.

```
contract TimelockStateRegistry is BaseStateRegistry, ITimelockStateRegistry, ReentrancyGuard {
    function processPayload(uint256 payloadId_) ... {
        ... // do all actions
        if (messageQuorum[_proof] < _getRequiredMessagingQuorum(srcChainId)) {
            revert Error.QUORUM_NOT_REACHED();
        }
    }
```

Recommendation: Move the Quorum check towards the beginning of the function.
Superform: Solved in PR 255.
Reviewer: Verified

#G-14 : Function `finalizePayload()` sets status on memory copy

Context: TimelockStateRegistry.sol#L109-L183
Description: Function `finalizePayload()` sets `p.status` to `TwoStepsStatus.PROCESSED` to prevent re-entrancy (according to the comment). However `p` is a memory copy of `timelockPayload[timeLockPayloadId_]` so a reentrant call won't notice this change.
Luckily the function `finalizePayload()` is also protected by `nonReentrant` so won't be an issue in practice.

```
contract TimelockStateRegistry is BaseStateRegistry, ITimelockStateRegistry, ReentrancyGuard {
    function finalizePayload(...) ... nonReentrant {
        TimelockPayload memory p = timelockPayload[timeLockPayloadId_];
        ...
        if (p.status != TwoStepsStatus.PENDING) {
            revert Error.INVALID_PAYLOAD_STATUS();
        }
        ...
        /// @dev set status here to prevent re-entrancy
        p.status = TwoStepsStatus.PROCESSED;
        ...
    }
}
```

Recommendation: Doublecheck the required functionality for reentrancy and remove unnessary functionality.
Superform: Solved in PR 294.
Reviewer: Verified

#G-15 : Flags in `_processMultiDeposit()` can be set more efficient

Context: CoreStateRegistry.sol#L713-L800
Description: The function `_processMultiDeposit()` sets the flags `fulfilment` and `errors` to `true` if they are `false`. Logically it is the same as directly setting the values to `true`, which saves a `jump` instruction.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _processMultiDeposit(...) ... {
        bool fulfilment;
        bool errors;
        for (uint256 i; i < numberOfVaults;) {
            ...
            if (!fulfilment) fulfilment = true;
            ...
            if (!errors) errors = true;
            ...
         }
    }
}
```

Recommendation: Consider directly setting the flag to true.

```
-   if (!fulfilment) fulfilment = true;
+   fulfilment = true;
    ...
-   if (!errors) errors = true;
+   errors = true;
```

Superform: Almost Solved in PR 295 and PR 320.
Reviewer: Verified

#G-16 : `_processMultiDeposit` can be optimized

Context: CoreStateRegistry.sol#L713-L800
Description: The function `_processMultiDeposit()` checks for each `vault` if there are enough tokens present. It could happen that close to the end of the loop it turns out there are insufficient tokens. Then every previous actions has to be undone, which wastes a lot of gas. Assuming frequently the same token is used, then the `balanceOf` of same tokens is called multiple times.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _processMultiDeposit(...) ... {
        for (uint256 i; i < numberOfVaults;) {
            underlying = IERC20(IBaseForm(superforms[i]).getVaultAsset());
            if (underlying.balanceOf(address(this)) >= multiVaultData.amounts[i]) {
                ...
                try IBaseForm(superforms[i]).xChainDepositIntoVault(...) ... { ... } catch { ... }
                ...
            } else {
                revert Error.BRIDGE_TOKENS_PENDING();
            }
        }
    }
}
```

Recommendation: Consider first checking there are enough tokens. This requires adding the `amounts[]` per type of token first.
Superform: The assumption of frequently the same token assumption may not be true all the time, as users can re-use single asset / can use multiple asset in a multi vault deposit. To achieve this, we'd need to sum

based on amounts per kind of underlying of the vaults which is expensive and then we've to do the validations (could be cumbersome). Also, this function is a keeper function, hence the keeper will do off-chain calls to make sure this transaction succeeds to not waste significant gas fees.
Reviewer: Acknowledged

#G-17 : Use unchecked in function `_updateMultiVaultDepositPayload`

Context: CoreStateRegistry.sol#L430-L509
Description: In function `_updateMultiVaultDepositPayload` the statement `++currLen` could be `unchecked` too. This would be more consistent with the other uses of `unchecked` in combination with counter increments.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function _updateMultiVaultDepositPayload(...) ... {
        ...
        uint256 currLen;
        for (uint256 i; i < arrLen;) {
            if (...) {
                ...
                ++currLen; // could be unchecked
            }
            unchecked {
                ++i;
            }
        }
    }
}
```

Recommendation: Consider changing the code to:

```
+   unchecked {
        ++currLen;
+   }
```

Superform: Solved in PR 277.
Reviewer: Verified

#G-18 : For loops don't always cache length

Context: SuperRegistry.sol#L126, SuperRegistry.sol#L155, PaymentHelper.sol#L574, PaymentHelper.sol#L602, BaseRouterImplementation.sol#L995, CoreStateRegistry.sol#L358
Description: Most `for` loops iterating over an array use a cached version for the length. However the following loops don't. Caching saves some gas and would be more consistent.

```
settings\SuperRegistry.sol
126:        for (uint256 i; i < bridgeId_.length;) {
155:        for (uint256 i; i < ambId_.length;) {

payments\PaymentHelper.sol
574:        for (uint256 i; i < req_.length;) {
602:        for (uint256 i; i < liqReq_.length;) {

BaseRouterImplementation.sol
995:            for (uint256 j; j < targets_.length;) {

crosschain-data\extensions\CoreStateRegistry.sol
358:        for (uint256 i; i < superformIds.length;) {
```

Recommendation: Consider caching the length.
Superform: Solved in PR 312.
Reviewer: Verified

#G-19 : Function `processPayload()` can be optimized

Context: CoreStateRegistry.sol#L188-L271
Description: Function `processPayload()` can be optimized to save some gas.

```
contract CoreStateRegistry is BaseStateRegistry, ICoreStateRegistry {
    function processPayload(uint256 payloadId_)
       ...
        uint8[] memory proofIds = proofAMB[v._proof];
        if (returnMessage.length > 0) {
            uint8[] memory ambIds = new uint8[](proofIds.length + 1);
             ...
            uint256 len = proofIds.length;
            for (uint256 i; i < len;) {
                ambIds[i + 1] = proofIds[i];
                ...
            }
        ...
        }
    ...
    }
}
```

Recommendation: Consider changing the code to:

```
-uint8[] memory proofIds = proofAMB[v._proof];
if (returnMessage.length > 0) {
+   uint8[] memory proofIds = proofAMB[v._proof];
+   uint256 len = proofIds.length;
-   uint8[] memory ambIds = new uint8[](proofIds.length + 1);
+   uint8[] memory ambIds = new uint8[](len + 1);
     ...
-   uint256 len = proofIds.length;
    for (uint256 i; i < len;) {
        ambIds[i + 1] = proofIds[i];
        ...
    }
}
```

Superform: Solved in PR 312.
Reviewer: Verified

#G-20 : `synthethicTokenId[superformId_]` is first stored and then read again

Context: SuperTransmuter.sol#L92-L125
Description: `synthethicTokenId[superformId_]` is first stored and then read again. This takes more gas than neccessary.

```
contract SuperTransmuter is ISuperTransmuter, Transmuter, StateSyncer {
    function registerTransmuter(uint256 superformId_, bytes memory extraData_) external override ret
        ...
        synthethicTokenId[superformId_] = address(new sERC20(...));
        ...
        return synthethicTokenId[superformId_];
    }
```

Recommendation: Consider storing the result of `address(new sERC20(...))` in a temporary variable and use that to assign `synthethicTokenId[superformId_]` and use it as return value.
Superform: Solved in PR 303.
Reviewer: Verified

#G-21 : Nested `if` instead of `&&`

Context: SuperPositions.sol#L128-L219, SuperTransmuter.sol#L197-L295
Description: The functions `stateMultiSync()` and `stateSync()` of `SuperPositions` and `SuperTransmuter` use a nested `if` to check two conditions. In other parts of code `&&` is used.

Although the nested if is slightly cheaper it is a different pattern.

```
function stateMultiSync(AMBMessage memory data_)
    ...
    if (callbackType != uint256(CallbackType.RETURN)) {
            if (callbackType != uint256(CallbackType.FAIL)) revert Error.INVALID_PAYLOAD();
        }
    ...
}
function stateSync(AMBMessage memory data_)
    ...
    if (callbackType != uint256(CallbackType.RETURN)) {
            if (callbackType != uint256(CallbackType.FAIL)) revert Error.INVALID_PAYLOAD();
        }
    ...
}
```

Recommendation: Preferably use the same coding pattern everywhere. On places where gas usage is off utmost important consider used nested `if`s instead of `&&`.
Superform: Solved in PR 302.
Reviewer: Verified

#G-22 : Constants `PRECISION_DECIMALS` and `PRECISION` not used

Context: BaseForm.sol#L27-L29
Description: The contract `BaseForm` defines the constants `PRECISION_DECIMALS` and `PRECISION` but they are not used.

```
abstract contract BaseForm is Initializable, ERC165Upgradeable, IBaseForm {
    ...
    uint256 internal constant PRECISION_DECIMALS = 27;
    uint256 internal constant = 10 ** PRECISION_DECIMALS;
    ...
}
```

Recommendation: Consider removing the constants `PRECISION_DECIMALS` and `PRECISION`.
Superform: Solved in PR 291.
Reviewer: Verified

#G-23 : Assignment of `v.permit2` can be done only when necessary

Context: BaseRouterImplementation.sol#L922-L1004
Description: In function `_multiVaultTokenForward()`, the value for `v.permit2` is assigned early on, and might not even be used.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _multiVaultTokenForward(...) ... {
        ...
        v.permit2 = superRegistry.PERMIT2();
        ...
        if (v.totalAmount > 0) {
            if (v.permit2dataLen > 0) {
                ...
                IPermit2(v.permit2).permitTransferFrom( ... )
                ...
            }
        }
    }
}
```

Recommendation: Consider moving the v.permit2 inside the `if (v.permit2dataLen > 0) ...`. This saves some gas and is consistent with `_singleVaultTokenForward()`.
Superform: Solved in PR 267.
Reviewer: Verified

#G-24 : Function `_multiVaultTokenForward()` evaluates `vaultData_.liqData[0].token` twice

Context: BaseRouterImplementation.sol#L922-L1004
Description: In `_multiVaultTokenForward()` the value of `vaultData_.liqData[0].token` is retrieved twice and the second time its stored in a temporary variable. It would save some gas to store it in the temporary variable the first time.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _multiVaultTokenForward(...) ... {
        if (vaultData_.liqData[0].token != NATIVE) {
            ...
            v.token = IERC20(vaultData_.liqData[0].token);
            ...
       }
    }
}
```

Recommendation: Consider changing the code to the example below. This also emphesises the special role for token 0.

```
function _multiVaultTokenForward(...) ... {
    address token = IERC20(vaultData_.liqData[0].token);
    if (token == NATIVE) {
        ...
    }
}
```

Superform: Solved in PR 278.
Reviewer: Verified

#G-25 : Inconsistent checks for `amount_ == 0`

Context: BaseRouterImplementation.sol#L572-L616, BaseRouterImplementation.sol#L669-L706, BaseRouterImplementation.sol#L922-L1004
Description: The function `_directDeposit()` checks for `amount_ == 0`, which means all of the local deposit functions have this check. However all `xchain` functions and `withdraw` functions don't have this check, which might waste gas for unnecessary actions.
Function `_multiVaultTokenForward()` has checks for `v.totalAmount == 0`. In that situation it could also skip the `safeIncreaseAllowance` or revert.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    function _directDeposit(...) ... {
        ...
        if (amount_ == 0) {
            revert Error.ZERO_AMOUNT();
        }
        ...
    }
   function _multiVaultTokenForward(
        ...
        for (uint256 i; i < v.len;) {
            ...
            v.totalAmount += v.approvalAmounts[i];
            ...
        }
        if (v.totalAmount > 0) { // could revert if v.totalAmount == 0
           ...
        }
        for (uint256 j; j < targets_.length;) {      // could skip this if v.totalAmount == 0
            v.token.safeIncreaseAllowance(targets_[j], v.approvalAmounts[j]);
            ...
        }
```

```
    }
}
```

Recommendation: Consider adding checks for `amount_ == 0` and determine the best locations. A possible location to do this is in the functions `_validateSuperformData()`, `_validateSuperformsDepositData()` and `_validateSuperformsWithdrawData()`. Also see issue Local deposits and withdraws don't check the paused state of forms .
Superform: Solved in PR 275.
Reviewer: Verified

#G-26 : Field `permit2` of struct `ValidateAndDispatchTokensArgs` is never used

Context: BaseRouterImplementation.sol#L414-L448
Description: The struct `ValidateAndDispatchTokensArgs` contains a field `permit2` that is never used in `_validateAndDispatchTokens()`.

```
abstract contract BaseRouterImplementation is IBaseRouterImplementation, BaseRouter, LiquidityHandle
    struct ValidateAndDispatchTokensArgs {
        ...
        address permit2;
        ...
    }
    function _validateAndDispatchTokens(ValidateAndDispatchTokensArgs memory args_) internal virtual
        ...
        IBridgeValidator(bridgeValidator).validateTxData(...); // doesn't use permit2
        dispatchTokens(...); // doesn't use permit2
    }
}
```

Recommendation: Consider removing the `permit2` field from struct `ValidateAndDispatchTokensArgs`.
Superform: Solved in PR 270.
Reviewer: Verified