CANTINA

Superform

Competition

February 13, 2024

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

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

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

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

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

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

From Now 27th to Dec 18th the Cantina team conducted a review of ERC1155A.The team identified a total of **214** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 4
- Medium Risk: 18
- Low Risk: 87
- Gas Optimizations: 15
- Informational: 90

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

3 Findings

3.1 High Risk

3.1.1 `Superpositions.onlyminter()` has wrong implemerantion, leading to wrong access control

Submitted by chaduke, also found by trachev

Severity: High Risk

Context: SuperPositions.sol#L69-L83

Description: `SuperPositions.onlyMinter()` has wrong implemerantion, leading to wrong access control. The problem is that the following implementation is comparing the `formImplementationId` instead of the state registry id for that superform to the state registry id of `msg.sender`.

The correct comparison is to retrieve the state registry id of the superform using function `getStateRegistryId()`, and then compare this id to the id of `msg.sender`.

Due to the wrong comparison, the implementation is wrong. As a result, any function uses this modifier will have the wrong access control, a serious security vulnerability.

A similar problem exists for the modifier `onlyBatchMinter()`.

Recommendation: Compare the correct state registry id as follows:

```
  modifier onlyMinter(uint256 superformId) {
      address router = superRegistry.getAddress(keccak256("SUPERFORM_ROUTER"));

      /// if msg.sender isn't superformRouter then it must be state registry for that superform
      if (msg.sender != router) {
-          (, uint32 formBeaconId,) = DataLib.getSuperform(superformId);
(address superform_, uint32 formImplementationId_, uint64 chainId_) = DataLib.getSuperform(superformId);

+        uint8 formRegistryId = ERC4626FormImplementation(superform).getStateRegistryId();

          uint8 registryId = superRegistry.getStateRegistryId(msg.sender);

-          if (uint32(registryId != formBeaconId) {
+          if (registryId != formRegistryId) {

              revert Error.NOT_MINTER();
          }
      }

      _;
  }
```

3.1.2 Incorrect allowance decrement in safebatchtransferfrom function

Submitted by Vijay, also found by Cosine, zigtur, 0x3b, ljmanini, Victor Okafor , Said, sashik-eth, pep7siup, XDZ-IBECX, hals, KupiaSec, 0xG0P1 and minhquanym

Severity: High Risk

Context: ERC1155A.sol#L140-L145

Description: `ERC1155A`'s `safeBatchTransferFrom`, instead of decrementing `msg.sender`'s allowance, the receiver's allowance is being decremented.

Proof of concept:

1. Let's say Alice has 100 super position tokens.
2. Alice gave 5 Tokens allowance to Bob.
3. Alice also gave infinite approval (`approve(type(uint256).max)`) to a smart contract X which can pull tokens from Alice when she calls a specific function.
4. Now Bob can use his 5 Tokens allowance to lock all the Alice's tokens by transferring all of them to the smart contract X by providing the smart contract address as to address.

Hence, a malicious allowee can lock all the tokens of the owner.

Recommenation:

Instead of decrementing receiver's allowance, decrement `msg.sender`'s allowance in `safeBatchTransferFrom` function.

```
  if (singleApproval) {
      if (allowance(from, msg.sender, id) < amount) revert NOT_ENOUGH_ALLOWANCE();
-     allowances[from][to][id] -= amount;
+     allowances[from][msg.sender][id] -= amount;
  }
```

3.1.3 Lack of refund handling from lifi bridge

Submitted by ladboy233 - Sparkware

Severity: High Risk

Context: BaseRouterImplementation.sol#L188

The protocol intends to use lifi to bridge token when there are cross chain deposits/withdrawals. However, as outlined in the lifi documentation,

> REFUNDED: The transfer was not successful and the sent token has been refunded
>
> When using amarok or hop it can happen that receiving.token is not the token requested in the original quote: amarok mints custom nextToken when bridging and swap them automatically to the token representation the user requested. In rare cases, it can happen that while the transfer was executed, the swap liquidity to exchange that token was used up. In this case, the user receives the nextToken instead. You can go to this webpage to exchange that token later.
>
> hop mints custom hToken when bridging and swap them automatically to the token representation the user requested. In rare cases, it can happen that while the transfer was executed, the swap liquidity to exchange that token was used up. In this case, the user receives the nextToken instead. You can go to this webpage to exchange that token later

there exists a possibility that a cross-chain transfer might fail, resulting in the token being refunded. However, there is lack of refund handling from lifi refund.

When the `msg.sender` is identified as the router during the call (here):

```
/// @dev dispatches tokens through the selected liquidity bridge to the destination contract
_dispatchTokens(
    superRegistry.getBridgeAddress(args_.liqRequest.bridgeId),
    args_.liqRequest.txData,
    args_.liqRequest.token,
    IBridgeValidator(bridgeValidator).decodeAmountIn(args_.liqRequest.txData, true),
    args_.liqRequest.nativeAmount
);
```

it implies that the refunded token from the Lifi bridge is likely to be lost.

Recommendation: Handle refunds when a user bridges funds via lifi

3.1.4 `ERC4626KYCDaoForm` contract cannot hold kycdDAO NFT which breaks the core

Submitted by 0xhuy0512, also found by ladboy233 - Sparkware and 0xarno

Severity: High Risk

Context: ERC4626KYCDaoForm.sol#L12, ERC4626KYCDaoForm.sol#L31

Description: The `ERC4626KYCDaoForm` contract must hold a kycDAO NFT, which in reality is impossible. There are 2 reasons why it is impossible for the contract to hold the NFT:

1. The kycDAO NFT is non-transferable and only accepts minting directly to the caller. There are no methods in `ERC4626KYCDaoForm` that can mint kycDAO NFT to itself.
2. `ERC4626KYCDaoForm` doesn't implement `ERC721Receiver`, which is required to hold kycDAO NFT.

The impact of not holding the NFT is that the `ERC4626KYCDaoForm` contract can't surpass the `kycDAO4626.hasKYC()` modifier, which prevents the contract from depositing or withdrawing from the vault, yielding the contract unusable.

Recommendation: Consider:

- Implementing a method that calls to kycDAO NFT's `mintWithCode()`.
- Implementing the `ERC721Receiver` standard.

3.2 Medium Risk

3.2.1 `retainerc4626` flag mishandling in `_updatemultideposit`

Submitted by elhaj

Severity: Medium Risk

Context: CoreStateRegistry.sol#L386

Description: After cross-chain multideposit arrives to the `coreStateRegistry`, the keeper will update the deposit first, before processing it through updateDepositPayload().

```
function updateDepositPayload(uint256 payloadId_, uint256[] calldata finalAmounts_) external virtual override {
    _onlyAllowedCaller(keccak256("CORE_STATE_REGISTRY_UPDATER_ROLE"));

    // some code ...

    PayloadState finalState;
    if (isMulti != 0) {
        // See the line below
            (prevPayloadBody, finalState) = _updateMultiDeposit(payloadId_, prevPayloadBody, finalAmounts_);
    } else {
        // this will may or may not update the amount n prevPayloadBody .
        (prevPayloadBody, finalState) = _updateSingleDeposit(payloadId_, prevPayloadBody, finalAmounts_[0]);
    }
    // some code ...
}
```

In this case, the `_updateMultiDeposit` function is responsible for updating the deposit payload by resolving the final amounts given by the keeper. In this process, the failed deposits will be removed from the payload body and set to `failedDeposit` to be rescued later by the user.

```
function _updateMultiDeposit(
    uint256 payloadId_,
    bytes memory prevPayloadBody_,
    uint256[] calldata finalAmounts_
    )
    internal
    returns (bytes memory newPayloadBody_, PayloadState finalState_)
    {
    /// some code ...

    uint256 validLen;
    for (uint256 i; i < arrLen; ++i) {
        if (finalAmounts_[i] == 0) {
            revert Error.ZERO_AMOUNT();
        }
        // update the amounts :
        (multiVaultData.amounts[i],, validLen) = _updateAmount(
            dstSwapper,
            multiVaultData.hasDstSwaps[i],
            payloadId_,
            i,
            finalAmounts_[i],
            multiVaultData.superformIds[i],
            multiVaultData.amounts[i],
            multiVaultData.maxSlippages[i],
            finalState_,
            validLen
        );
    }
    // update the payload body and remove the failed deposits
    if (validLen != 0) {
```

```
        uint256[] memory finalSuperformIds = new uint256[](validLen);
        uint256[] memory finalAmounts = new uint256[](validLen);
        uint256[] memory maxSlippage = new uint256[](validLen);
        bool[] memory hasDstSwaps = new bool[](validLen);

        uint256 currLen;
        for (uint256 i; i < arrLen; ++i) {
            if (multiVaultData.amounts[i] != 0) {
                finalSuperformIds[currLen] = multiVaultData.superformIds[i];
                finalAmounts[currLen] = multiVaultData.amounts[i];
                maxSlippage[currLen] = multiVaultData.maxSlippages[i];
                hasDstSwaps[currLen] = multiVaultData.hasDstSwaps[i];
                ++currLen;
            }
        }
        multiVaultData.amounts = finalAmounts;
        multiVaultData.superformIds = finalSuperformIds;
        multiVaultData.maxSlippages = maxSlippage;
        multiVaultData.hasDstSwaps = hasDstSwaps;
        finalState_ = PayloadState.UPDATED;
    } else {
        finalState_ = PayloadState.PROCESSED;
    }
    // return new payload
    newPayloadBody_ = abi.encode(multiVaultData);
}
```

The provlem arises when some deposits fail and others succeed. The function doesn't update the `retainERC4626` flags to match the new state:

This misalignment can lead to incorrect minting behavior in the `_multiDeposit` function, where the `retainERC4626` flags do not correspond to the correct `superFormsIds`.

Proof of concept:

1. Bob creates a `singleXchainMultiDeposit` with its corresponding data:
   - `superFormsIds[1,2,3,4]`
   - `amounts[a,b,c,d]`
   - `retainERC4626[true,true,false,false]`
2. After the cross-chain payload is received and all is good, a keeper comes and updates the amounts. Assuming the update of amounts resulted in `a` and `b` failing, while `c` and `d` are resolved successfully.
3. The `_updateMultiDeposit` function updates the payload to contain:
   - `superFormsIds[3,4]`
   - `amounts[c',d']`
   - `retainERC4626[true,true,false,false]`
3. Here, `retainERC4626` is not updated. So, when the keeper processes the payload, `_multiDeposit` is triggered, and Bob is incorrectly minted superPositions for superForms 3 and 4, despite the user's preference to retain `ERC4626` shares for these superForms.

Therefore, the issue can lead to incorrect minting or unminting behavior of superPositions, where users may receive `superPositions` when they intended to retain ERC4626 shares, or vice versa. While it may be not a big issue for EOAs, this can be particularly problematic for contracts integrating with Superform, potentially breaking invariants and causing loss of funds.

Recommenation: The `retainERC4626` flags should be realigned with the updated `superFormsIds` and `amounts` to ensure consistent behavior. A new array for `retainERC4626` should be constructed in parallel with the other arrays to maintain the correct association.

Here is a suggested fix:

```
// Inside the _updateMultiDeposit function
bool[] memory finalRetainERC4626 = new bool[](validLen);
// ... existing code to update superFormsIds and amounts ...

uint256 currLen;
for (uint256 i; i < arrLen; ++i) {
    if (multiVaultData.amounts[i] != 0) {
        // ... existing code to update finalSuperformIds and finalAmounts ...
        finalRetainERC4626[currLen] = multiVaultData.retainERC4626[i];
        ++currLen;
    }
}

// Update the multiVaultData with the new retainERC4626 array
multiVaultData.retainERC4626 = finalRetainERC4626;
```

3.2.2 Attacker can deposit to non existing superform and change the token that should be returned

Submitted by rvierdiiev, also found by elhaj

Severity: Medium Risk

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

To deposit on another chain, users must call `SuperformRouter.singleXChainSingleVaultDeposit` as option with single vault deposit. Then `BaseRouterImplementation._singleXChainSingleVaultDeposit` will be called.

The User provides `SingleXChainSingleVaultStateReq` param to the function were he can provide all info. One of such info is `superformId`, which is information about superform vault and chain where it is located.

`SuperformRouter.singleXChainSingleVaultDeposit` function then validates that superfom data. In the case of a direct deposit or withdrawal (on the same chain), then the function checks if a Superform exist. In our case, this check will be skipped as we deposit to another chain.

Then it checks that superform chain is the same as the destination chain. And after that, it is checked that the superform implementation is not paused.

The function `_validateSuperformData` doesn't have ability to check if superform indeed exists on destination.

`_singleXChainSingleVaultDeposit` doesn't have any additional checks for the superform, so it packs superform and then sends it to the destination. This means that a user can provide any superform id as a deposit target.

In case of cross-chain deposits, all funds should go to the dst swapper or core registry. This is forced by validators (both lifi and socket). Sp in case if you don't do dst swap, then funds will be bridged directly to the `CoreStateRegistry` on destination chain.

When the message is relayed and proofs are relayed, then keepers can call `updateDepositPayload` and provide amount that was received from user after bridging. And only after that, `processPayload` can be called. `updateDepositPayload` function will call `_updateSingleDeposit` to update the amount that is received on destination chain, then `_updateAmount` will be called.

In case there was no dstSwap (and also in case that slippage didn't pass) then superform is checked to be valid. In our case, if we provided not valid `superformId`, then we also enter that logic branch.

- CoreStateRegistry.sol#L561-L562:

```
failedDeposits[payloadId_].superformIds.push(superformId_);

address asset;
try IBaseForm(_getSuperform(superformId_)).getVaultAsset() returns (address asset_) {
    asset = asset_;
} catch {
    /// @dev if its error, we just consider asset as zero address
}
/// @dev if superform is invalid, try catch will fail and asset pushed is address (0)
/// @notice this means that if a user tries to game the protocol with an invalid superformId, the funds
/// bridged over that failed will be stuck here
failedDeposits[payloadId_].settlementToken.push(asset);
failedDeposits[payloadId_].settleFromDstSwapper.push(false);

/// @dev sets amount to zero and will mark the payload as PROCESSED (overriding the previous memory
/// settings)
amount_ = 0;
finalState_ = PayloadState.PROCESSED;
```

Inside this block, the payload is marked as failed and the call to the superform is done to get assets in it. In the case that the vault has provided assets, it will be set to the `failedDeposits[payloadId_].settlementToken`. As our superform vault is invalid and can be crafted by an attacker, it can be crafted in such way that it provides more precious token than the one that was deposited by the attacker.

Later, the failed payload will be resolved using `proposeRescueFailedDeposits` and the payload initiator will be refunded. In case they do not notice that token has changed, it is possible that they refund the wrong amount of tokens. This is also a problem in case the attacker compromises the `CORE_STATE_REGISTRY_RESCUER_ROLE`.

Therefore, an attacker can get bigger amount of funds.

Recommendation: In the case of an invalid superform, then don't update `failedDeposits[payloadId_].settlementToken.push(asset)` to the token returned by invalid vault.

3.2.3 `Erc4626formimplementation._processdirectwithdraw` sends funds to `msg.sender` instead of receiver

Submitted by rvierdiiev

Severity: Medium Risk

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

Proof of concept: When a user wants to do a withdrawalon the same chain (or on another chain), then they provide the `receiverAddress` of the funds inside their request.

When the request is executed in the `ERC4626FormImplementation._processDirectWithdraw` function, the `srcSender` parameter is used instead. As result, funds will be withdrawn to the wrong address, which can be problematic, especially for some other vaults that are constructed on top of superform vaults and should send funds to another recipient.

_Note that for withdrawals on another chain, funds are sent to the correct receiver address.

Suppose that another contract uses superform as its vault and holds all superpositions inside. Users receive shares when depositing into it. And in order to execute a withdrawal, that contract will pass the user's address as the receiver. However, funds will be sent back to the contract and it's likely that the user will get nothing. However for cross-chain it will work fine, so it's possible that they will not notice that quickly.

As result, such a contract will not be able to work normally and some user's will lose funds: funds can be sent to another recipient, which can lead to loss of funds.

Recommendation: Send funds to the `receiverAddress` only. Also make sure to use it here as well.

3.2.4 Withdrawal on xchain will revert when srcsender != receiveraddress

Submitted by rvierdiiev

Severity: Medium Risk

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

Proof of concept: When a user wants to perform a withdrawal, then they provide the `receiverAddress` of the funds inside their request. It's possible that the user does not provide `txData` in the `liqRequest` and will wait for keepers to do that for them on the destination chain.

This can be done using `CoreStateRegistry.updateWithdrawPayload` and eventually, it will call `CoreStateRegistry._updateTxData`. So, keeper will provide `txData` using this function and this `txData` will be validated. Note that `srcSender` is passed to it, which will be used to check that the same address is used in the `txData` for the receiver.

So, to make the update not fail, keeper should provide source chain initiators as fund receivers in `txData`. Otherwise, the function will revert.

After this is done, then cross-chain withdrawal can be processed. Inside `ERC4626FormImplementation._processXChainWithdraw`, the `txData` provided by keeper is checked again, and now `singleVaultData_.receiverAddress` is passed as `srcSender`, which means that, in case of the source chain sender not being equal to the `singleVaultData_.receiverAddress`, validation will revert and the withdrawal will not be executed (i.e. it will revert).

Recommendation: Use `singleVaultData_.receiverAddress` inside `CoreStateRegistry._updateTxData` to check if the provided `txData` is valid.

3.2.5 Keeper can steal funds from the dstswapper

Submitted by rvierdiiev

Severity: Medium Risk

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

Proof of concept: When a user deposits to another chain, then they can do a dst swap, which means that after bridging from source to destination, the bridge wont' send the vault's asset to the destination chain. So this interim token should be swapped to the vault's asset, which is done by keepers by calling `DstSwapper.processTx` function and provide swapping `txData`, which contains info about the swap.

All validation and swaps are done inside `_processTx`. The token and amount that should be sent are then decoded from txData and token is checked to be same as interim token of the user. After that, `txData` is validated, to check that the receiver of the swap is set to the CSR. Then the swap is done.

In the end, there is a check that CSR indeed has increased the balance with the amount that user has requested.

Keepers currently belong to the protocol, however they have plans to change this and allow other entities to execute that role.

A malicious keeper can create small deposit from one chain to another with dst swap and set the interim token to the one that they want to steal. In this deposit, they canl provide high (> 90) slippage to sandwich a the subsequent swap (that swaps the entire balance of the interim token in the DstSwapper) parametrized in the provided a `txData` in order to get profit from the high slippage and large amount of tokens.

As result, after the swap a certain amount of tokens will still arrive to the CSR, explaining why the balance and slippage checks pass and the transaction wont revert, while allowing the malicious keeper to steal user's funds from the DstSwapper.

Recommendation: Consider implementing some kind of oracles in order to calculate the approximate value of the tokens that the depositor expects to receive, and then disallow the keeper to use more than that value in the other asset $\pm$ some deviation.

3.2.6 Vaults write function return values could not reflect the real output

Submitted by solthodox, also found by ladboy233 - Sparkware, ljmanini and ethan

Severity: Medium Risk

Context: ERC4626FormImplementation.sol#L240

Description: When the superform calls either the `deposit` or `redeem` methods of the underlying vault, it uses the return value of those functions to fetch the amount of shares minted on `deposit` and the amount of assets withdrawn on `redeem`. Some vaults could not reflect the reality in their return values (maybe they decided to return an empty value, or they return the share values when redeeming, instead of the real assets, which might be less due to slippage of swaps in the withdraw process) and this would make the superform have an incorrect accounting of its shares and superpositions, potentially accounting users' positions unfairly.

Recommendation: Fetch the real output directly from the shares or assets balance, using `balanceOf` for extra security. The contract could even require the return value to equal the real output for more sanity.

```
if (singleVaultData_.retain4626) {
    uint256 balanceBefore = v.balanceOf(address(singleVaultData_.receiverAddress));
    v.deposit(vars.assetDifference, singleVaultData_.receiverAddress);
    dstAmount = v.balanceOf(address(singleVaultData_.receiverAddress)) - balanceBefore;
} else {
//...
```

3.2.7 Keeper will always overwrite the user `txData` in case of single cross-chain withdrawal

Submitted by elhaj, also found by Said

Severity: Medium Risk

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

Description: The keeper will always ovewrite the `txData` when updating the withdrawal payload in case of a single withdrawal, while it shouldn't be updated if the user already provided a `txData` like in multi-withdrawal behavior.

The `updateWithdrawPayload` function within the `CoreStateRegistry` contract plays the role of updating the withdrawal payload with a (`txData_`) provided by the keeper. Particularly, in the _updateWithdrawPayload function:

```
// updateWithdrawPayload function
// ...
// see the line below
prevPayloadBody = _updateWithdrawPayload(prevPayloadBody, srcSender, srcChainId, txData_, isMulti);
// more code ..
```

This function is expected to call _updateTxData to conditionally update the payload with the keeper's `txData_`:

```
// _updateWithdrawPayload() function
// ...
// see the line below
multiVaultData = _updateTxData(txData_, multiVaultData, srcSender_, srcChainId_, CHAIN_ID);
// more code ..
```

The _updateTxData function should leave the user's `txData` unchanged if it is already provided (i.e. if its length is not zero):

```
function _updateTxData(/*....*/) internal view returns (InitMultiVaultData memory){

    uint256 len = multiVaultData_.liqData.length;

    for (uint256 i; i < len; ++i) {
        // see the line below
        if (txData_[i].length != 0 && multiVaultData_.liqData[i].txData.length == 0) {
            // ...
        }
    }

    return multiVaultData_;
}
```

In case of `singleWithdraw`, regardless of whether the user's `txData` was provided or not, the function will always overwrite the user's `singleVaultData.liqData.txData` to `txData_[0]`, which is the keeper's data, instead of the `txData` returned from _updateTxData, which will not be the keeper `txData` in case user provided data:

```
function _updateWithdrawPayload( bytes memory prevPayloadBody_, address srcSender_, uint64 srcChainId_,
bytes[] calldata txData_,uint8 multi )internal  view returns (bytes memory){
    // prev code ..
    multiVaultData = _updateTxData(txData_, multiVaultData, srcSender_, srcChainId_, CHAIN_ID);

    if (multi == 0) {
        // @audit-issue : the keeper will always overwrite the txData,should be
↪   multiVaultData.liqData.txdata[0]
        // see the line below
        singleVaultData.liqData.txData = txData_[0];
        return abi.encode(singleVaultData);
    }

    return abi.encode(multiVaultData);
}
```

The correct behavior should ensure that the user's `txData` is preserved when provided (like in multi-withdrawal), and the keeper's `txData` is only employed if the user's ‘txData“ is absent.

The impact however is somewhat unclear in this scenario because of the unknown keeper behavior. Nevertheless, due to the validation process prior to executing the `txData`, there are some potential issues in case of a maliciously `txData` passing the validation:

1. The `amountIn` for the swap can be set to an undesirably low value, causing the withdrawn amounts of the user to swap only a small portion, with the remainder staying in the superForm.
2. The final received token after the swap to any token can be altered (e.g. swapping from USDC to WETH, the keeper might set it to swap from USDC to any Token).
3. The behavior of the `txData` can be changed (e.g. from swap and bridge, to only swap).

Recommenation:

```
  // prev code ...
  if (multi == 0) {
-     singleVaultData.liqData.txData = txData_[0];
+     singleVaultData.liqData.txData = multiVaultData.liqData.txdata[0]
      return abi.encode(singleVaultData);
  }
  // more code ..
```

3.2.8 `directdepositintovault` of superforms should check max slippage provided by user

Submitted by Said, also found by Nyksx and ethan

Severity: Medium Risk

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

Description: When users want to deposit to superforms directly in the same chain, they can call `singleDirectSingleVaultDeposit` or `singleDirectMultiVaultDeposit` (see SuperformRouter.sol#L25-L34)

```
/// @inheritdoc IBaseRouter
function singleDirectSingleVaultDeposit(SingleDirectSingleVaultStateReq memory req_)
    external
    payable
    override(BaseRouter, IBaseRouter)
{
    uint256 balanceBefore = address(this).balance - msg.value;

    _singleDirectSingleVaultDeposit(req_);
    _forwardPayment(balanceBefore);
}
```

Also see SuperformRouter.sol#L49-L58:

```
function singleDirectMultiVaultDeposit(SingleDirectMultiVaultStateReq memory req_)
    external
    payable
    override(BaseRouter, IBaseRouter)
{
    uint256 balanceBefore = address(this).balance - msg.value;

    _singleDirectMultiVaultDeposit(req_);
    _forwardPayment(balanceBefore);
}
```

In these functions, user can provide max slippage as parameters and will be passed to superform when trigger `directDepositIntoVault`.

```
struct SingleDirectSingleVaultStateReq {
    SingleVaultSFData superformData;
}

struct SingleVaultSFData {
    // superformids must have same destination. Can have different underlyings
    uint256 superformId;
    uint256 amount;
    uint256 maxSlippage;
    LiqRequest liqRequest; // if length = 1; amount = sum(amounts)| else  amounts must match the amounts being
↪  sent
    bytes permit2data;
    bool hasDstSwap;
    bool retain4626; // if true, we don't mint SuperPositions, and send the 4626 back to the user instead
    address receiverAddress;
    bytes extraFormData; // extraFormData
}

struct SingleDirectMultiVaultStateReq {
    MultiVaultSFData superformData;
}

struct MultiVaultSFData {
    // superformids must have same destination. Can have different underlyings
    uint256[] superformIds;
    uint256[] amounts;
    uint256[] maxSlippages;
    LiqRequest[] liqRequests; // if length = 1; amount = sum(amounts) | else  amounts must match the amounts
↪  being sent
    bytes permit2data;
    bool[] hasDstSwaps;
    bool[] retain4626s; // if true, we don't mint SuperPositions, and send the 4626 back to the user instead
    address receiverAddress;
    bytes extraFormData; // extraFormData
}
```

Also see BaseRouterImplementation.sol#L556-L589:

```
/// @notice fulfils the final stage of same chain deposit action
function _directDeposit(
    address superform_,
    uint256 payloadId_,
    uint256 superformId_,
    uint256 amount_,
    uint256 maxSlippage_,
    bool retain4626_,
    LiqRequest memory liqData_,
    address receiverAddress_,
    bytes memory extraFormData_,
    uint256 msgValue_,
    address srcSender_
)
    internal
    virtual
    returns (uint256 dstAmount)
{
    /// @dev deposits token to a given vault and mint vault positions directly through the form
    dstAmount = IBaseForm(superform_).directDepositIntoVault{ value: msgValue_ }(
        InitSingleVaultData(
            payloadId_,
            superformId_,
            amount_,
            maxSlippage_, // @audit - maxSlippage is provided and passed to superform
            liqData_,
            false,
            retain4626_,
            receiverAddress_,
            /// needed if user if keeping 4626
            extraFormData_
        ),
        srcSender_
    );
}
```

However, when `directDepositIntoVault` is called, the `maxSlippage` is never checked when processing the token amount. Instead, it strictly checks that `vars.assetDifference` is not lower than `singleVaultData_.amount` (see ERC4626FormImplementation.sol#L231-L233:

```
function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (uint256
↪  dstAmount) {
    directDepositLocalVars memory vars;

    IERC4626 v = IERC4626(vault);
    vars.asset = address(asset);
    vars.balanceBefore = IERC20(vars.asset).balanceOf(address(this));
    IERC20 token = IERC20(singleVaultData_.liqData.token);

    if (address(token) != NATIVE && singleVaultData_.liqData.txData.length == 0) {
        /// @dev this is only valid if token == asset (no txData)
        if (singleVaultData_.liqData.token != vars.asset) revert Error.DIFFERENT_TOKENS();

        /// @dev handles the asset token transfers.
        if (token.allowance(msg.sender, address(this)) < singleVaultData_.amount) {
            revert Error.DIRECT_DEPOSIT_INSUFFICIENT_ALLOWANCE();
        }

        /// @dev transfers input token, which is the same as vault asset, to the form
        token.safeTransferFrom(msg.sender, address(this), singleVaultData_.amount);
    }

    /// @dev non empty txData means there is a swap needed before depositing (input asset not the same as vault
    /// asset)
    if (singleVaultData_.liqData.txData.length != 0) {
        vars.bridgeValidator = superRegistry.getBridgeValidator(singleVaultData_.liqData.bridgeId);

        vars.chainId = CHAIN_ID;

        vars.inputAmount =
            IBridgeValidator(vars.bridgeValidator).decodeAmountIn(singleVaultData_.liqData.txData, false);

        if (address(token) != NATIVE) {
```

```
            /// @dev checks the allowance before transfer from router
            if (token.allowance(msg.sender, address(this)) < vars.inputAmount) {
                revert Error.DIRECT_DEPOSIT_INSUFFICIENT_ALLOWANCE();
            }

            /// @dev transfers input token, which is different from the vault asset, to the form
            token.safeTransferFrom(msg.sender, address(this), vars.inputAmount);
        }

        IBridgeValidator(vars.bridgeValidator).validateTxData(
            IBridgeValidator.ValidateTxDataArgs(
                singleVaultData_.liqData.txData,
                vars.chainId,
                vars.chainId,
                vars.chainId,
                true,
                address(this),
                msg.sender,
                address(token),
                address(0)
            )
        );

        _dispatchTokens(
            superRegistry.getBridgeAddress(singleVaultData_.liqData.bridgeId),
            singleVaultData_.liqData.txData,
            address(token),
            vars.inputAmount,
            singleVaultData_.liqData.nativeAmount
        );

        if (
            IBridgeValidator(vars.bridgeValidator).decodeSwapOutputToken(singleVaultData_.liqData.txData)
                != vars.asset
        ) {
            revert Error.DIFFERENT_TOKENS();
        }
    }

    vars.assetDifference = IERC20(vars.asset).balanceOf(address(this)) - vars.balanceBefore;

    /// @dev the difference in vault tokens, ready to be deposited, is compared with the amount inscribed in
↪   the
    /// superform data
    // see the line below
    if (vars.assetDifference < singleVaultData_.amount) {
        revert Error.DIRECT_DEPOSIT_INVALID_DATA();
    }

    /// @dev notice that vars.assetDifference is deposited regardless if txData exists or not
    /// @dev this presumes no dust is left in the superform
    IERC20(vars.asset).safeIncreaseAllowance(vault, vars.assetDifference);

    if (singleVaultData_.retain4626) {
        dstAmount = v.deposit(vars.assetDifference, singleVaultData_.receiverAddress);
    } else {
        dstAmount = v.deposit(vars.assetDifference, address(this));
    }
}
```

This could lead to an issue, especially if the users use an allowed swap provider, most likely asset will not be exactly the same and slightly less than the provided `singleVaultData_.amount`. The calls could result in a revert most of the time.

Recommendation: Check the user's provided `maxSlippage` and check that `vars.assetDifference` must not be lower than the allowed slippage instead:

```
  function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (uint256
↪   dstAmount) {
      directDepositLocalVars memory vars;

      IERC4626 v = IERC4626(vault);
      vars.asset = address(asset);
      vars.balanceBefore = IERC20(vars.asset).balanceOf(address(this));
      IERC20 token = IERC20(singleVaultData_.liqData.token);
```

```
    if (address(token) != NATIVE && singleVaultData_.liqData.txData.length == 0) {
        /// @dev this is only valid if token == asset (no txData)
        if (singleVaultData_.liqData.token != vars.asset) revert Error.DIFFERENT_TOKENS();

        /// @dev handles the asset token transfers.
        if (token.allowance(msg.sender, address(this)) < singleVaultData_.amount) {
            revert Error.DIRECT_DEPOSIT_INSUFFICIENT_ALLOWANCE();
        }

        /// @dev transfers input token, which is the same as vault asset, to the form
        token.safeTransferFrom(msg.sender, address(this), singleVaultData_.amount);
    }

    /// @dev non empty txData means there is a swap needed before depositing (input asset not the same as
↪   vault
    /// asset)
    if (singleVaultData_.liqData.txData.length != 0) {
        vars.bridgeValidator = superRegistry.getBridgeValidator(singleVaultData_.liqData.bridgeId);

        vars.chainId = CHAIN_ID;

        vars.inputAmount =
            IBridgeValidator(vars.bridgeValidator).decodeAmountIn(singleVaultData_.liqData.txData, false);

        if (address(token) != NATIVE) {
            /// @dev checks the allowance before transfer from router
            if (token.allowance(msg.sender, address(this)) < vars.inputAmount) {
                revert Error.DIRECT_DEPOSIT_INSUFFICIENT_ALLOWANCE();
            }

            /// @dev transfers input token, which is different from the vault asset, to the form
            token.safeTransferFrom(msg.sender, address(this), vars.inputAmount);
        }

        IBridgeValidator(vars.bridgeValidator).validateTxData(
            IBridgeValidator.ValidateTxDataArgs(
                singleVaultData_.liqData.txData,
                vars.chainId,
                vars.chainId,
                vars.chainId,
                true,
                address(this),
                msg.sender,
                address(token),
                address(0)
            )
        );

        _dispatchTokens(
            superRegistry.getBridgeAddress(singleVaultData_.liqData.bridgeId),
            singleVaultData_.liqData.txData,
            address(token),
            vars.inputAmount,
            singleVaultData_.liqData.nativeAmount
        );

        if (
            IBridgeValidator(vars.bridgeValidator).decodeSwapOutputToken(singleVaultData_.liqData.txData)
                != vars.asset
        ) {
            revert Error.DIFFERENT_TOKENS();
        }
    }

    vars.assetDifference = IERC20(vars.asset).balanceOf(address(this)) - vars.balanceBefore;

    /// @dev the difference in vault tokens, ready to be deposited, is compared with the amount inscribed in
↪   the
    /// superform data
-   if (vars.assetDifference < singleVaultData_.amount) {
-       revert Error.DIRECT_DEPOSIT_INVALID_DATA();
-   }
+   if (vars.assetDifference < ((singleVaultData_.amount * (10_000 - singleVaultData_.maxSlippage)) /
↪   10_000)) {
+       revert Error.DIRECT_DEPOSIT_INVALID_DATA();
```

```
+     }

      /// @dev notice that vars.assetDifference is deposited regardless if txData exists or not
      /// @dev this presumes no dust is left in the superform
      IERC20(vars.asset).safeIncreaseAllowance(vault, vars.assetDifference);

      if (singleVaultData_.retain4626) {
         dstAmount = v.deposit(vars.assetDifference, singleVaultData_.receiverAddress);
      } else {
          dstAmount = v.deposit(vars.assetDifference, address(this));
      }
  }
```

3.2.9 `Lifivalidator.validatetx` performs an incorrect receiver check for lifi over stargate/celer/amarok bridging

Submitted by cergyk

Severity: Medium Risk

Context: LiFiValidator.sol#L48

Description: LiFi is a liquidity bridge which acts as a wrapper on top of many bridges. Among them is Stargate, which itself is built upon the messaging mechanism of Layerzero. When calling LiFi over Stargate for dispatching tokens, Superform does not check the receiver address correctly, which means that a malicious user could send tokens to themselves and, depending on off-chains mechanisms of Superform, they can attempt to use the liquidity bridged by another user for themselves as well.

Let's examine the `LiFiValidator.validateTxData` function:

```
function validateTxData(ValidateTxDataArgs calldata args_) external view override returns (bool hasDstSwap) {
    /// @dev xchain actions can have bridgeData or bridgeData + swapData
    /// @dev direct actions with deposit, cannot have bridge data - goes into catch block
    /// @dev withdraw actions may have bridge data after withdrawing - goes into try block
    /// @dev withdraw actions without bridge data (just swap) - goes into catch block

    try this.extractMainParameters(args_.txData) returns (
        string memory, /*bridge*/
        address sendingAssetId,
        address receiver, /* see this line */
        uint256, /*amount*/
        uint256, /*minAmount*/
        uint256 destinationChainId,
        bool, /*hasSourceSwaps*/
        bool hasDestinationCall
    ) {
        // ...
    } catch {
        // ...
    }
}
```

We see that the receiver extracted is the one from the `ILiFi.BridgeData` struct, and checks are carried out solely on that value. However, during the call to `StargateFacet`, the value of `_stargateData.callTo` is used as the receiver on target chain:

```
function _startBridge(
    ILiFi.BridgeData memory _bridgeData,
    StargateData calldata _stargateData
) private {
    if (LibAsset.isNativeAsset(_bridgeData.sendingAssetId)) {
        //Native token handling
        // ...
    } else {
        LibAsset.maxApproveERC20(
            IERC20(_bridgeData.sendingAssetId),
            address(composer),
            _bridgeData.minAmount
        );

        composer.swap{ value: _stargateData.lzFee }(
            getLayerZeroChainId(_bridgeData.destinationChainId),
            _stargateData.srcPoolId,
            _stargateData.dstPoolId,
            _stargateData.refundAddress,
            _bridgeData.minAmount,
            _stargateData.minAmountLD,
            IStargateRouter.lzTxObj(
                _stargateData.dstGasForCall,
                0,
                toBytes(address(0))
            ),
            _stargateData.callTo,
            _stargateData.callData
        );
    }

    emit PartnerSwap(0x0006);

    emit LiFiTransferStarted(_bridgeData);
}
```

There is a security check during the call which checks that `callData` is empty when `!_bridgeData.hasDestinationCall` (enforced during `validateTxData`) in StargateFacet.sol#L119 and StargateFacet.sol#L229-L239.

But it is still possible to use this call with empty data, and tokens are still transferred to `_stargateData.callTo` on the destination chain.

Therefore, a user can attempt to steal the liquidity on the destination chain provided by another user (if the keeper sees the bridging action as completed and uses the liquidity already available in `DstSwapper`), or other liquidity already present in `DstSwapper`. The exact impact depends on checks carried out by the offchain logic of keepers.

Recommenation: Take a look at the additional checks carried out in CalldataVerificationFacet.sol#L210-L302 to see how additional parameters should be checked when calling these underlying bridges

3.2.10 `Dstswapper.processtx` an attacker can use multiple destination swaps back and forth to inflate xchain deposit

Submitted by cergyk

Severity: Medium Risk

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

Description: A malicious user can use multiple back and forth swaps to inflate the amount of deposit

The operator calling `DstSwapper` can process swaps in batch for a given `payloadId`. If the payload contains multiple swaps between tokenA and tokenB for amount X, the liquidity the sender needs to provide initially is only X tokenA.

However, as all swaps are finished, there are N successful swaps each for X in token A and token B. This means that if otherwise the liquidity is available in `DstSwapper`, the sender can take it.

This is due to the fact that `DstSwapper` only checks the target token increase during a swap, but does not register its decrease when used in another swap for the same `payloadId`.

```
function _processTx(
    uint256 payloadId_,
    uint256 index_,
    uint8 bridgeId_,
    bytes calldata txData_,
    address userSuppliedInterimToken_,
    IBaseStateRegistry coreStateRegistry_
)
    internal
{
    // Unrelated validation logic
    ...

    /// @dev get the address of the bridge to send the txData to.
    (v.underlying, v.expectedAmount, v.maxSlippage) = _getFormUnderlyingFrom(coreStateRegistry_, payloadId_,
↪  index_);

    // See the line below
    v.balanceBefore = IERC20(v.underlying).balanceOf(v.finalDst);

    // Swap tokens in bridge
    // ...
    // See the line below
    v.balanceAfter = IERC20(v.underlying).balanceOf(v.finalDst);

    if (v.balanceAfter <= v.balanceBefore) {
        revert Error.INVALID_SWAP_OUTPUT();
    }
    // See the line below
    v.balanceDiff = v.balanceAfter - v.balanceBefore;

    /// @dev if actual underlying is less than expAmount adjusted
    /// with maxSlippage, invariant breaks
    /// @notice that unlike in CoreStateRegistry slippage check inside updateDeposit, in here we don't check
↪  for
    /// negative slippage
    /// @notice this essentially allows any amount to be swapped, (the invariant will still break if the
↪  amount is
    /// too low)
    /// @notice this doesn't mean that the keeper or the user can swap any amount, because of the 2nd slippage
↪  check
    /// in CoreStateRegistry
    /// @notice in this check, we check if there is negative slippage, for which case, the user is capped to
↪  receive
    /// the v.expAmount of tokens (originally defined)
    if (v.balanceDiff < ((v.expectedAmount * (10_000 - v.maxSlippage)) / 10_000)) {
        revert Error.SLIPPAGE_OUT_OF_BOUNDS();
    }

    /// @dev updates swapped amount
    // See the line below
    swappedAmount[payloadId_][index_] = v.balanceDiff;

    /// @dev emits final event
    emit SwapProcessed(payloadId_, index_, bridgeId_, v.balanceDiff);
}
```

This strategy is conditioned on the fact that the malicious user Alice can get the operator to carry the swaps on the destination chain even though the liquidity of Alice has not been bridged correctly on destination chain.

As such, this vulnerability is dependent on how off-chain infrastructures handle failed token bridging, but here are some strategies to get the bridging of tokens to fail intentionally:

- Supply unmatched deposit tokens/interim tokens for cross-chain deposit.
- Bridge tokens to a receiver address controlled by the attacker.
- Use gas parameter manipulation to make the call fail on destination chain.

Therefore, a malicious user can steal liquidity contained in `DstSwapper` which, depending on off-chain infra can be provided by other users.

Recommenation: Two solutions may be envisioned:

- Register pulled balances of tokens during swaps.
- Impose that a target token in a swap cannot be a source token in another swap for the same `payloadId`.

3.2.11 A malicious user can craft valid calldata to call 'packed' version of some of lifi endpoints

Submitted by cergyk

Severity: Medium Risk

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

Description: Some endpoints of the LiFi bridge do not conform to the shape `(ILiFi.BridgeData, /*...*/)`. In that case it is possible to craft a calldata which decodes to the tuple `(ILiFi.BridgeData, /*...*/)`, but is also a valid input to the endpoint. In that case, `validateTxData` does not validate the right data, and the malicious user ends up bridging liquidity to themselves.

This is possible because some endpoints on the LiFi diamond enable `packing` of calldata:

- HopFacetPacked:
  - `startBridgeTokensViaHopL2NativePacked`
  - `startBridgeTokensViaHopL2NativeMin`
- CBridgeFacetPacked:
  - `startBridgeTokensViaCBridgeNativePacked`
  - `startBridgeTokensViaCBridgeNativeMin`
  - `startBridgeTokensViaCBridgeERC20Packed`
  - `startBridgeTokensViaCBridgeERC20Min`

Let's take the example of `CBridgeFacetPacked.startBridgeTokensViaCBridgeERC20Packed`:

```
function startBridgeTokensViaCBridgeERC20Min(
    bytes32 transactionId,
    address receiver,
    uint64 destinationChainId,
    address sendingAssetId,
    uint256 amount,
    uint64 nonce,
    uint32 maxSlippage
) external;
```

We see that it expects a calldata of the shape: `(bytes32, address, uint64, address, uint256, uint64, uint32)`, which is static (no dynamic types). On the other hand, `ILiFi.BridgeData` is a dynamic struct, which means that the first 32 bytes word holds the offset at which the struct is located.

In the case of the calldata for the packed endpoint, the first 32 bytes word is `bytes32 transactionId`, which is an arbitrary blob used for tracking/analytics. So we can easily craft a calldata which decodes validly for both cases:

```
struct BridgeData {
    bytes32 transactionId;
    string bridge;
    string integrator;
    address referrer;
    address sendingAssetId;
    address receiver;
    uint256 minAmount;
    uint256 destinationChainId;
    bool hasSourceSwaps;
    bool hasDestinationCall;
}

function testEncodeCollision() public {
    //This variable acts as the transactionId during the actual call,
    //But also works as the offset for the encoding of the dynamic struct
    bytes32 transactionId = bytes32(uint(0x100));
```

```
    //Completely different receiver
    address receiver = address(1337);

    uint64 destinationChainId = 1;
    address sendingAssetId = address(3);
    uint256 amount = 1000;
    uint64 nonce = 1001;
    uint32 maxSlippage = 1002;

    bytes memory _calldata = abi.encode(
        transactionId,
        receiver,
        destinationChainId,
        sendingAssetId,
        amount,
        nonce,
        maxSlippage
    );

    BridgeData memory bridgeData = BridgeData(
        bytes32(uint(1)),
        'stargate',
        'jumper.exchange',
        address(1),
        address(2),

        //This is the receiver checked by `validateTxData`
        address(3),
        4,
        1,
        false,
        false
    );

    bytes memory bridgeDataEncoded = abi.encode(bridgeData);

    // We concat both encodings
    _calldata = abi.encodePacked(_calldata, bridgeDataEncoded);


    // First we can decode the format needed for the endpoint
    (, address actualReceiver, , , , ,) = abi.decode(_calldata, (
        bytes32,
        address,
        uint64,
        address,
        uint256,
        uint64,
        uint32
    ));

    // But we can also decode the format needed for validateTx
    BridgeData memory bridgeDataDecoded = abi.decode(_calldata, (
        BridgeData
    ));

    assert(bridgeDataDecoded.receiver == bridgeData.receiver);
    assert(actualReceiver == address(1337));

    console.logBytes(_calldata);
}
```

A malicious user can use this to bypass `validateTx` checks, and send tokens to themselves on the destination chain. Depending on the mechanism implemented off-chain, it can be tricked into using liquidity already available in `DstSwapper`, since the bridging will succeed.

Recommenation: Two solutions are possible:

- Blacklist the `packed` endpoints.
- Force the calldata to have the `BridgeData` at the start (offset `0x20`), making this collision impossible in practice.

3.2.12 Lifivalidator.validatetxdata calldata can decode to bridgedata type but also be compatible with generic swap

Submitted by cergyk

Severity: Medium Risk

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

Description: `LiFiValidator.validateTxData` uses the assumption that if the calldata is valid for the generic swap function call, it will fail to decode during `extractMainParameters`. This is why the cross-chain case is handled in a `try` block, whereas local in `catch` block.

However, it is possible to create a calldata which is validly decoded by `extractMainParameters`, and also valid when calling `swapTokensGeneric` on the LiFi diamond.

Furthermore, the sets of parameters that these encodings represent do not overlap, which means a malicious user can entirely bypass the validation logic, and use an arbitrary local swap when indicating a cross-chain deposit to superform

Note that in this case `swapTokensGeneric`, which has the signature:

```
function swapTokensGeneric(
    bytes32 _transactionId,
    string calldata _integrator,
    string calldata _referrer,
    address payable _receiver,
    uint256 _minAmount,
    LibSwap.SwapData[] calldata _swapData
) external payable
```

has the same arbitrary blob `_transactionId` which can be used to specify an offset for the location of the dynamic type `BridgeData`.

Therefore, a malicious user can use this to bypass the `validateTx` chain entirely, and execute a local swap when indicating a cross-chain deposit to superform.

Recommenation: Match on the selector called, instead of relying on encoding matching (which can be collision prone)

3.2.13 Malicious admin combing with low-level call can steal funds from user or form

Submitted by ladboy233 - Sparkware

Severity: Medium Risk

Context: LiquidityHandler.sol#L54

Description: The intended functionality of `_dispatchTokens` is to enable users to bridge tokens or conduct token swaps through the 1inch exchange, utilizing low-level calls (see LiquidityHandler.sol#L33):

```
function _dispatchTokens(
    address bridge_,
    bytes memory txData_,
    address token_,
    uint256 amount_,
    uint256 nativeAmount_
)
    internal
    virtual
{
    if (bridge_ == address(0)) {
        revert Error.ZERO_ADDRESS();
    }

    if (token_ != NATIVE) {
        IERC20 token = IERC20(token_);
        token.safeIncreaseAllowance(bridge_, amount_);
    } else {
        if (nativeAmount_ < amount_) revert Error.INSUFFICIENT_NATIVE_AMOUNT();
    }

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

Address bridge is expected to be either Lifi address or 1inch address.This function is usually called in this way:

```
/// @dev dispatches tokens through the selected liquidity bridge to the destination contract
_dispatchTokens(
    superRegistry.getBridgeAddress(args_.liqRequest.bridgeId),
    args_.liqRequest.txData,
    args_.liqRequest.token,
    IBridgeValidator(bridgeValidator).decodeAmountIn(args_.liqRequest.txData, true),
    args_.liqRequest.nativeAmount
);
```

However, malicious admin can whitelist a malicious address so the query

```
superRegistry.getBridgeAddress(args_.liqRequest.bridgeId)
```

can return a token address. Consider the case:

1. A user grants infinite spending allowance to the router for UDSC token.
2. Admin is compromised and a hacker whitelists the USDC token address for bridge id 1000. So,

   ```
   superRegistry.getBridgeAddress(args_.liqRequest.bridgeId)
   ```

   returns the USDC address
3. The hacker craft the following payload data:

   ```
   abi.encodeWithSelector(IERC20.transferFrom.selector, address(victim), address(hacker), userUSDCBalance)
   ```

4. Then the hacker can steal the USDC that sits in users wallet by using the following low-level call:

   ```
   (bool success,) = payable(bridge_).call{ value: nativeAmount_ }(txData_);
   ```

Recommenation: Do not use low-level calls, use explicit calls with interface function instead and prevent the admin from whitelist token address.

3.2.14 When `retain4626` is `false`, the superposition may be minted to an incorrect address

Submitted by Nyksx

Severity: Medium Risk

Context: BaseRouterImplementation.sol#L593-L630

Description: When depositing to the vault, users can make deposits on behalf of other users by specifying the receiver address. If the user doesn't retain 4626, the function will mint super positions at the end of the deposit action.

```
if (dstAmount != 0 && !vaultData_.retain4626) {
    /// @dev mint super positions at the end of the deposit action if user doesn't retain 4626
    ISuperPositions(superRegistry.getAddress(keccak256("SUPER_POSITIONS"))).mintSingle(
        srcSender_, vaultData_.superformId, dstAmount
    );
}
```

The issue occurs when the user wants to deposit to another address but doesn't want to keep 4626, resulting in the super position being minted for the `msg.sender` instead of the receiver address.

Recommendation: Mint superposition for the receiver's address when the function caller wants to deposit to a different address and does not want to keep 4626.

3.2.15 Vaults for assets that can rebase negatively are prone to unexpectedly revert

Submitted by ljmanini, also found by ladboy233 - Sparkware

Severity: Medium Risk

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

Description: Superform is designed to allow any external protocol to integrate their yield-generating vault into the system by wrapping it within a `Form` contract. As of the current audit, only `ERC4626` vaults are supported.

Superform enables teams to integrate their vault into the system and leverage its integrations with cross-chain bridging solutions, providing users on any supported chain access to the external team's product.

However, the cross-chain withdrawal process can be subject to unexpected reverting transactions on the withdrawal's destination chain, especially when a vault employs an asset that may negatively rebase, similar to how Pods Finance's stETHvv Vault functions in the case of Lido suffering a slashing event.

The problem lies in the strictness of the check employed within the `ERC4626FormImplementation#_processXChainWithdraw()` method.

This function calls `dstAmount = vault.redeem()` to withdraw the user's funds, and in the case where `singleVaultData_.liqData.txData.length != 0`, it compares `dstAmount` with the withdraw payload's `txData`-stored `amount`, a value written during the payload's update process.

If the vault's asset suffers a negative rebase between the payload's update and its processing, which presumably occurs as two separate transactions, the amount of asset represented by the user's shares may reduce below the expected `amount`, causing the check at ERC4626FormImplementation.sol#L367 to unexpectedly revert.

Proof of concept:

1. Assume Alice currently has a cross-chain deposit, done via Superform, into a vault as described in this issue.
2. Alice initiates a cross-chain withdrawal, utilizing all her shares.
3. The cross-chain withdrawal payload is correctly processed by the AMBs, and a quorum of proofs reaches the Superform system on the destination chain.
4. The `UPDATER` keeper updates the payload via `CoreStateRegistry`, inserting a `txData` whose `amount` field is `X`.
5. The Vault's collateral asset suffers a 1% negative rebase: all holders now hold 1% less asset, which translates into its shares now being worth 99% of their initial value, in asset terms.

6. The `PROCESSOR` keeper will process the payload:
    - The transaction's flow reaches `ERC4626FormImplementation#_processXChainWithdraw()`, which redeems the vault's shares.
    - The vault returns 99% of the expected amount of asset because of the negative rebase.
    - As a consequence of the above, the transaction will revert, as `vars.amount > dstAmount`, i.e. the amount expected to be received is in fact greater than the amount of asset sent by the vault.

Recommenation: Consider adding the possibility for users to specify a tolerable slippage amount (or percentage) for their withdrawals: the current functionality can be maintained by setting such a parameter to `0`, while for a case such as the one presented, a non-`0` value may be recommended to the user.

As such, the amount of asset obtained from the Vault during the redeem process must be allowed to be above or equal to the expected amount, reduced by the user's tolerable slippage amount.

3.2.16 Superform protocol doesn't support vaults if their underlying asset is a fee-on-transfer type token

Submitted by hals, also found by pks271, 8olidity and Mario Poneder

Severity: Medium Risk

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

Description: The protocol allows any vault owner from adding their vaults to the protocol by wrapping them to one of the approved form implementations.

These vaults can have any type of underlying assets including fee-on-transfer token type; which deducts a fee from the transferred amount, so the resulting final balance of the receiver would be less than the sent amount by the amount of the deducted fees.

It was noticed that the protocol doesn't support such type of tokens; users can't deposit in vaults with fee-on-transfer tokens due to this check made in the `ERC4626FormImplementation._processDirectDeposit` function

```
vars.assetDifference = IERC20(vars.asset).balanceOf(address(this)) - vars.balanceBefore;

/// @dev the difference in vault tokens, ready to be deposited, is compared with the amount inscribed in the
/// superform data
if (vars.assetDifference < singleVaultData_.amount) {
    revert Error.DIRECT_DEPOSIT_INVALID_DATA();
}
```

where `vars.assetDifference` will always be less than `singleVaultData_.amount` due to the deducted fees (for fee-on-transfer underlying vault token).

Recommenation: Update the function to support fee-on-transfer tokens.

3.2.17 Increasing quorum requirements will prevent messages from being processed

Submitted by cccz, also found by elhaj, rvierdiiev and bronzepickaxe

Severity: Medium Risk

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

Description: The admin can call `setRequiredMessagingQuorum` to set `requiredQuorum`:

```
function setRequiredMessagingQuorum(uint64 srcChainId_, uint256 quorum_) external override onlyProtocolAdmin {
    requiredQuorum[srcChainId_] = quorum_;

    emit QuorumSet(srcChainId_, quorum_);
}
```

And the message can only be processed if the number of proof messages received is greater than the `requiredQuorum`:

```
if (messageQuorum[payloadProof] < _getQuorum(srcChainId)) {
    revert Error.INSUFFICIENT_QUORUM();
}
```

The problem here is that when a cross-chain message is initiated, the length of its `ambIds` is fixed, i.e. the number of messages sent is fixed, which also means that the number of received proof messages, i.e. `messageQuorum`, will not exceed it.

If `setRequiredMessagingQuorum` is called to increase the `requiredQuorum` between the initiation and processing of a cross-chain message, then when the cross-chain message is processed, it will not be processed due to insufficient `messageQuorum`:

1. Consider chain A with `requiredQuorum` 3, a user initiates a cross-chain deposit with ambIds length 4.
2. When the message is successfully dispatched, admin calls `setRequiredMessagingQuorum` to increase the `requiredQuorum` to 5.
3. When the destination chain processes the message, the message will not be processed because the `messageQuorum` is at most 4 and less than 5.

Recommendation: Consider caching the current quorum requirements in the payload when a cross-chain message is initiated, and using it instead of the latest quorum requirements for the checks.

3.2.18 Lack of user control over share allocation in vault deposits may lead to unpredictable outcomes

Submitted by elhaj, also found by hals, solthodox and Shaheen

Severity: Medium Risk

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

Description: The protocol lacks validation to ensure that the number of shares minted by a vault upon deposit aligns with the user's expectations, and the users have no choice but to accept any amount of shares resulted when depositing to a vault. This is particularly concerning for vaults that implement additional deposit strategies, which could be susceptible to front-running or other market manipulations.

For example, a vault might use the deposited assets to engage in yield farming activities, where the number of shares minted to the depositor is dependent on the current state of the external protocol. If this protocol's conditions change rapidly (such as through front-running), the current protocol design exposes the users to front running attacks.

See the following example from the `_processDirectDeposit` function of a superForm:

```
function _processDirectDeposit(InitSingleVaultData memory singleVaultData_) internal returns (uint256
↪  dstAmount) {
    // prev code ....
    if (singleVaultData_.retain4626) {
        //@audit : user can't refuse the amount of shares minted even if it's zero .
        // see the line below
        dstAmount = v.deposit(vars.assetDifference, singleVaultData_.receiverAddress);
    } else {
            // see the line below
            dstAmount = v.deposit(vars.assetDifference, address(this));
    }
}
```

```
function _directSingleDeposit( address srcSender_, bytes memory permit2data_,InitSingleVaultData memory
↪  vaultData_) internal virtual {
    // prev code ...
    // @audit : the contract mint any amount resulted from depositing , and the user have no control of that
    // see the line below
    if (dstAmount != 0 && !vaultData_.retain4626) {
        /// @dev mint super positions at the end of the deposit action if user doesn't retain 4626
        ISuperPositions(superRegistry.getAddress(keccak256("SUPER_POSITIONS"))).mintSingle(
            srcSender_, vaultData_.superformId, dstAmount
        );
    }
}
```

This could lead to users receiving fewer shares than anticipated if the vault's share price is affected by other on-chain activities.

Recommenation: To mitigate this risk, the user should have the ability to specify his desired `mintedAmount` of shares when they are depositing to a `superForm`.