Superform Core Security Review

Reviewer
Hans

September 2, 2023

Contents

1 Executive Summary

Over the course of 15 business days in total, Superform engaged with Hans to review superform-core.

Summary

Type of Project	Yield Marketplace
Timeline	14th Aug, 2023 - 1st Sep, 2023
Methods	Manual Review
Documentation	High
Testing Coverage	High

A comprehensive security review identified a total of 23 issues and 5 gas optimization suggestions.

Repository	**Initial Commit**
Superform	772f7e438c5c1b0c458216c6935cdf45c643ecc0

Total Issues

High Risk	2
Medium Risk	14
Low Risk	5
Informational	2
Gas Optimization	5

The reported vulnerabilities were addressed by the Superform team, and the mitigation underwent a review process and was verified by Hans.

Repository	**Final Commit**
Superform	TBD

2 Scope of the Audit

Everything in the `src` folder except `vendor` is in scope.

3 About Hans

Hans is an esteemed security analyst in the realm of smart contracts, boasting a firm grounding in mathematics that has sharpened his logical abilities and critical thinking skills. These attributes have fast-tracked his journey to the peak of the Code4rena leaderboard, marking him as the number one auditor in a record span of time. In addition to his auditor role, he also serves as a judge on the same platform. Hans' innovative insight is evident in his creation of Solodit, a vital resource for navigating consolidated security reports. In addition, he is a co-founder of Cyfrin, where he is dedicated to enhancing the security of the blockchain ecosystem through continuous efforts.

4 Disclaimer

I endeavor to meticulously identify as many vulnerabilities as possible within the designated time frame; however, I must emphasize that I cannot accept liability for any findings that are not explicitly documented herein. It is essential to note that my security audit should not be construed as an endorsement of the underlying business or product. The audit was conducted within a specified timeframe, with a sole focus on evaluating the security aspects of the solidity implementation of the contracts.

While I have exerted utmost effort in this process, I must stress that I cannot guarantee absolute security. It is a well-recognized fact that no system can be deemed completely impervious to vulnerabilities, regardless of the level of scrutiny applied.

5 Protocol Summary

Superform is a non-custodial yield marketplace. For DeFi protocols, it acts as an instant distribution platform for ERC4626-compliant vaults. For users, it allows them to interact with/ deposit into any opportunity on the platform from the chain and asset of their choice in a single transaction.

6 Additional Comments

The protocol heavily relies on off-chain mechanisms with multiple roles and I could not verify the correctness of the off-chain logic.

7 Findings

7.1 High Risk

7.1.1 Wrong validation of `srcAddress` in `lzReceive()`

Severity: High

Context: LayerzeroImplementation.sol#L163

Description: `lzReceive()` validates if `srcAddress_` is same as the trusted remote.

```
        bytes memory trustedRemote = trustedRemoteLookup[srcChainId_];
        if (srcAddress_.length != trustedRemote.length && keccak256(srcAddress_) !=
↪   keccak256(trustedRemote)) {
            revert Error.INVALID_SRC_SENDER();
        }
```

But it uses `&&` instead of `||` so it will pass most addresses of the same length.

Impact `lzReceive()` would be manipulated due to the wrong validation.

Recommendation: We should change `&&` to `||`.

```
        bytes memory trustedRemote = trustedRemoteLookup[srcChainId_];
        if (srcAddress_.length != trustedRemote.length || keccak256(srcAddress_) !=
↪   keccak256(trustedRemote)) {
            revert Error.INVALID_SRC_SENDER();
        }
```

Superform:

Hans:

7.1.2 First depositor can steal funds of others

Severity: High

Context: BaseForm.sol#L32

Description: SuperForm uses an ERC4626 vault inside and users can deposit/withdraw funds through the vault. But with the default ERC4626 implementation, the first depositor would be frontrun by an attacker like the below scenario.

1. Alice wants to deposit `2M * 1e6 USDC` to a pool.
2. Bob observes Alice's transaction, frontruns to deposit 1 wei `USDC` to mint 1 wei share, and transfers `1 M * 1e6 USDC` to the pool.
3. Alice's transaction is executed, since `totalAsset = 1M * 1e6 + 1` and `totalSupply = 1`, Alice receives `2M * 1e6 * 1 / (1M * 1e6 + 1) = 1` share.
4. The pool now has `3M*1e6 +1` assets and distributed 2 shares. Bob profits 0.5 M and Alice loses 0.5 M USDC.

Impact Users might lose funds by the first depositor.

Recommendation: For the ERC4626 vault, it's recommended to send the first min liquidity LP tokens to the zero address to enable share dilution when the vault has 0 `totalSupply`.

In this protocol, as the users can set the vault directly during the initialization, we should add more validations to prevent the above scenario.

Superform:

Hans:

7.2 Medium Risk

7.2.1 Malicious users can break the quorum logic using duplicate `ambIds`

Severity: Medium

Context: BaseStateRegistry.sol#L158

Description: According to the documentation, `Superform utilizes a multi-bridge system where one bridge sends the full payload and at least one other bridge sends a keccak-encoded proof.` And `_dispatchProof()` is used to send the proof.

```
    function _dispatchProof(
        address srcSender_,
        uint8[] memory ambIds_,
        uint64 dstChainId_,
        uint256[] memory gasToPay_,
        bytes memory message_,
        bytes[] memory overrideData_
    ) internal {
        AMBMessage memory data = abi.decode(message_, (AMBMessage));
        data.params = abi.encode(keccak256(message_));

        /// @dev i starts from 1 since 0 is primary amb id which dispatches the message itself
        for (uint8 i = 1; i < ambIds_.length; ) { //@audit ambIds[1] = ambIds[2]?
            uint8 tempAmbId = ambIds_[I];
            ...
            /// @dev proof is dispatched in the form of a payload
            tempImpl.dispatchPayload{value: gasToPay_[i]}(srcSender_, dstChainId_, abi.encode(data),
↪   overrideData_[i]);

            unchecked {
                ++i;
            }
        }
    }
```

But it doesn't check if all ids are unique and the below scenario would be possible.

1. Assume there are 4 AMBs and `requiredQuorum` of the chain is 2.
2. A malicious users call `dispatchPayload()` with `ambIds[3] = {amb0, amb1, amb1}` though `SuperformRouter`.
3. Then `_dispatchProof()` will send the proof to `amb1` twice.
4. When we check lzEndpoint, it increases `nonce` for each message and this validation won't revert for the same messages.
5. As a result, it will increase `messageQuorum[proofHash]` by 2 and they can bypass the quorum requirement.

Impact The quorum logic will be broken.

Recommendation: `_dispatchProof()` should validate all AMD ids are unique.

Superform:

Hans:

7.2.2 Lack of gas amount validation in `CelerImplementation.broadcastPayload()`

Severity: Medium

Context: CelerImplementation.sol#L109

Description: `broadcastPayload()` is used to send a payload to all configured destination chains.

```
/// @dev calculates the exact fee needed
        uint256 feesReq = messageBus.calcFee(message_);
        feesReq = feesReq * totalChains;

        for (uint64 i; i < totalChains; ) {
            uint64 chainId = broadcastChains[i];

            messageBus.sendMessage{value: d.gasPerDst[i]}(authorizedImpl[chainId], chainId, message_);
↪   //@audit might use a different gas amount

            unchecked {
                ++i;
            }
        }

        /// Refund unused fees
        /// NOTE: check security implications here
        (bool success, ) = payable(srcSender_).call{value: msg.value - feesReq}("");
```

It uses `d.gasPerDst[i]` for each broadcast but charges `feesReq` that is calculated using `estimateFees()` and it might charge more or less gas from `srcSender` if these amounts are different.

Impact `CelerImplementation` might charge a different amount of gas from `srcSender`.

Recommendation: `broadcastPayload()` should charge the same amount of gas from `srcSender` as the sum of `d.gasPerDst`.

Superform:

Hans:

7.2.3 `broadcastChains` might contain duplicate `ambChainIds`

Severity: Medium

Context: CelerImplementation.sol#L137

HyperlaneImplementation.sol#L127

LayerzeroImplementation.sol#L122

Description: The protocol admin can add new chain IDs using `setChainId()`.

```
    function setChainId(uint64 superChainId_, uint64 ambChainId_) external onlyProtocolAdmin {
        if (superChainId_ == 0 || ambChainId_ == 0) {
            revert Error.INVALID_CHAIN_ID();
        }

        ambChainId[superChainId_] = ambChainId_;
        superChainId[ambChainId_] = superChainId_;

        broadcastChains.push(ambChainId_); //@audit might contain dups

        emit ChainAdded(superChainId_);
    }
```

This function might be called several times when the admin wants to update the `(superChainId, ambChainId)` mapping. In this case, `broadcastChains` will contain duplicate `ambChainIds` and there is no option to remove them.

Impact `broadcastPayload()` might broadcast a message several times to one chain.

Recommendation: We should add an option to remove the old elements or verify that `broadcastChains` doesn't already contain the `ambChainId`.

Superform:

Hans:

7.2.4 `setChainId()` should reset the old `ambChainId/superChainId` mappings.

Severity: Medium

Context: CelerImplementation.sol#L134

HyperlaneImplementation.sol#L123

LayerzeroImplementation.sol#L118

Description: The protocol admin can add `superChainId/ambChainId` using `setChainId()`.

```
    function setChainId(uint64 superChainId_, uint64 ambChainId_) external onlyProtocolAdmin {
        if (superChainId_ == 0 || ambChainId_ == 0) {
            revert Error.INVALID_CHAIN_ID();
        }

        ambChainId[superChainId_] = ambChainId_; //@audit should reset old mappings
        superChainId[ambChainId_] = superChainId_;

        broadcastChains.push(ambChainId_);

        emit ChainAdded(superChainId_);
    }
```

But it doesn't reset the old mapping and the below scenario would be possible.

1. The admin called `setChainId(1, 1)`. Then `ambChainId[1] = 1, superChainId[1] = 1`
2. After that, let's assume `setChainId(2, 1)` is called as the admin wants to update the mapping.
3. Then `ambChainId[2] = 1, superChainId[1] = 2` but `ambChainId[1]` is still 1.
4. So in dispatchPayload(), it might work with the old `chainId` when it shouldn't work.
5. If the admin calls `setChainId(1, 2)` instead, `superChainId[1]` will keep the old mapping and executeMessage() would be affected.

Impact Some functions including `dispatchPayload()` and `executeMessage()` might work with the old mappings.

Recommendation: `setChainId()` should remove the old mappings.

```
    function setChainId(uint64 superChainId_, uint64 ambChainId_) external onlyProtocolAdmin {
        if (superChainId_ == 0 || ambChainId_ == 0) {
            revert Error.INVALID_CHAIN_ID();
        }

        // reset old mappings
        uint64 oldSuperChainId = superChainId[ambChainId_];
        uint64 oldAmbChainId = ambChainId[superChainId_];

        if (oldSuperChainId > 0) {
            ambChainId[oldSuperChainId] = 0;
        }

        if (oldAmbChainId > 0) {
            superChainId[oldAmbChainId] = 0;
        }

        // set new mappings
        ambChainId[superChainId_] = ambChainId_;
        superChainId[ambChainId_] = superChainId_;

        broadcastChains.push(ambChainId_);

        emit ChainAdded(superChainId_);
    }
```

Superform:

Hans:

7.2.5 Gas fee calculation for AMB data transfer is incorrect

Severity: Medium

Context: SuperformRouter#L589-L618

Description: For any cross-chain deposits, there are two AMB messages transferred:

- Main message from `chainA` to `chainB` carrying cross-chain deposit information (aka `msgA`)
- ACK message from `chainB` to `chainA` when deposit is successful which is likely to happen always (aka `msgB`)

However, the protocol only charges their users the gas fees for `msgA`, which includes gas fees for users to execute tx on `chainA` plus for the tx processor on `chainB` to process the payload. It does not include gas fees for `msgA`, which includes gas fees for the tx processor on `chainB` to send the ACK message plus for the tx processor on `chainA` to process the payload.

An ACK message has 160 bytes of length which will approximately will consume 80K~90K gas on Ethereum, which isn't minor.

Impact Users would need to pay more gas using `PayMaster` as they are required to send less gas during the initial action.

Recommendation: We should calculate the gas fees properly.

Superform:

Hans:

7.2.6 Setting the quorum of updated messages to the required quorum causes `processPayload` to revert when the required quorum increases

Severity: Medium

Context: CoreStateRegistry#L96 CoreStateRegistry#L142

Description: In `updateDepositPayload` and `updateWithdrawPayload` functions, it updates the payload with appropriate token amounts so that the new payload can be processed through `processPayload`. However, at the end of updating the payload, it sets the message quorum of the new payload to the required quorum. This causes correct transactions to be reverted in `processPayload` if the required quorum is increased because of a security issue or whatever.

Here's an example:

1. Assume there are 10 AMBs out there and the required quorum is set to 6.
2. A new message arrived and says its quorum is 9.
3. Update action will be executed by an updater, generate a new message, and the new message quorum is set to 6.
4. If the required quorum is increased to 7 because of security issues of cross-chain protocols, all these updated transactions will revert in `processPayload` because it's now checking if the message quorum is equal to or greater than 7.

Impact Correct messages might not be transferred and executed.

Recommendation: Set the quorum of the new message with the quorum of the original message as follows:

```
uint256 originalQuorum = messageQuorum[v.prevPayloadProof];

...

messageQuorum[
    keccak256(abi.encode(AMBMessage(v.prevPayloadHeader, newPayloadBody)))
] = originalQuorum;
```

Superform:

Hans:

7.2.7 Single point of failure on the assumption of the same deployed addresses using CREATE2

Severity: Medium

Context: PayMaster.sol#L86

Description: We assume all multiTxProcessor instances are at the same address across chains.

```
    /// @inheritdoc IPayMaster
    function rebalanceToMultiTxProcessor(LiqRequest memory req_) external onlyPaymentAdmin {
        /// assuming all multi-tx processor across chains should be same; CREATE2
        address receiver = superRegistry.multiTxProcessor();

        if (receiver == address(0)) {
            revert Error.ZERO_ADDRESS();
        }

        _validateAndDispatchTokens(req_, receiver);
    }
```

Here's another example. src/payments/PaymentHelper.sol

```
    function _estimateSwapFees(
        uint64 dstChainId_,
        LiqRequest[] memory liqReq_
    ) internal view returns (uint256 gasUsed) {
        uint256 totalSwaps;

        for (uint256 i; i < liqReq_.length; ) {
            /// @dev checks if tx_data receiver is multiTxProcessor
            if (
                liqReq_[i].bridgeId != 0 &&
↪   IBridgeValidator(superRegistry.getBridgeValidator(liqReq_[i].bridgeId)).validateReceiver(
                    liqReq_[i].txData,
                    superRegistry.multiTxProcessor() // @audit issue
                )
```

However, this is cumbersome to handle. Here's why: CREATE2 address derivation is determined by the deployer contract address, salt, and contract creation code (contract byte code + constructor arguments).

1. MultiTxProcessor is related to SuperRegistry address, SuperRegistry is related to SuperRBAC, SuperRBAC is related to the admin address. These addresses should be all the same across chains. Of course, admin can be changed after creation, but still it needs extra care. Any mistake during the course needs to be restarted from scratch.
2. In some parts of the project, chainId is used to retrieve multiTxProcessor address, contrary to the assumption and this will lead to difficulty in maintenance. src/crosschain-liquidity/socket/SocketValidator.sol

```
    /// @inheritdoc BridgeValidator
    function validateTxData( //@audit-info validate txData
        bytes calldata txData_,
        uint64 srcChainId_,
        uint64 dstChainId_,
        bool deposit_,
        address superform_,
        address srcSender_,
        address liqDataToken_
    ) external view override {
        ISocketRegistry.UserRequest memory userRequest = _decodeCallData(txData_);

        /// @dev 1. chainId validation
        if (uint256(dstChainId_) != userRequest.toChainId) revert Error.INVALID_TXDATA_CHAIN_ID();

        /// @dev 2. receiver address validation
        if (deposit_) { //@audit-info validate for deposit
            if (srcChainId_ == dstChainId_) {
                /// @dev If same chain deposits then receiver address must be the superform
                if (userRequest.receiverAddress != superform_) revert Error.INVALID_TXDATA_RECEIVER();
            } else {
                /// @dev if cross chain deposits, then receiver address must be the token bank
                if (
                    !(userRequest.receiverAddress ==
↪   superRegistry.coreStateRegistryCrossChain(dstChainId_) ||
                        userRequest.receiverAddress ==
↪   superRegistry.multiTxProcessorCrossChain(dstChainId_))
                ) revert Error.INVALID_TXDATA_RECEIVER();
```

Impact If the assumption is broken, PayMaster and PaymentHelper will stop functioning leading to break of fee handling.

Recommendation: Reconsider the assumption and make changes accordingly.

Superform:

Hans:

7.2.8 Insufficient validation of token/amount can lead to the stealing of protocol funds via xchain deposits and withdrawal

Severity: Medium

Context: SuperformRouter.sol#L523

Description: In `_buildDepositAmbData` during building amb message data, `superformData_.amount` is set to the amount field of `InitSingleVaultData`. It's also checked against the amount inside `superformData_.liqRequest.txData`.

```
    /// @dev internal function used for validation and ambData building across different entry points
    function _buildDepositAmbData(
        uint64 dstChainId_,
        SingleVaultSFData memory superformData_
    ) internal returns (InitSingleVaultData memory ambData, uint256 currentPayloadId) {
        /// @dev validate superformsData
        if (!_validateSuperformData(dstChainId_, superformData_)) revert
 ↪  Error.INVALID_SUPERFORMS_DATA();

        if (
            !IBridgeValidator(superRegistry.getBridgeValidator(superformData_.liqRequest.bridgeId))
                .validateTxDataAmount(superformData_.liqRequest.txData, superformData_.amount)
        ) revert Error.INVALID_TXDATA_AMOUNTS();

        currentPayloadId = ++payloadIds;
        LiqRequest memory emptyRequest;

        ambData = InitSingleVaultData(
            currentPayloadId,
            superformData_.superformId,
            superformData_.amount,
            superformData_.maxSlippage,
            emptyRequest,
            superformData_.extraFormData
        );
    }
```

The route cause is missing validation of the input token against the superform asset.

Because the protocol relies on the off-chain mechanism heavily, it is recommended to add as much validation on-chain as possible.

Impact Malicious actors can exploit this vulnerability by inputting a fake token and pretending to be a real asset. The protocol will then use other users' funds (real assets) to deposit and mint the real super positions to the exploiter.

Recommendation: Specify srcToken and dstToken and check dstToken against the given superform asset.

Superform:

Hans:

7.2.9 Insufficient validation for Chainlink Oracle price data can affect fee calculation leading to protocol or user funds loss

Severity: Medium

Context: PaymentHelper.sol#L659 PaymentHelper.sol#L670

Description: The protocol is using Chainlink Oracle to determine the prices of native assets.

Specifically, `latestRoundData()` is used which is good since it's recommended from Chainlink.

```
function latestRoundData() external view
    returns (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    )
```

However the implementation details miss some important validations against return values. e.g.

```
    /// @dev helps return the current gas price of different networks
    /// @dev returns default set values if an oracle is not configured for the network
    function _getGasPrice(uint64 chainId_) internal view returns (uint256) {
        if (address(gasPriceOracle[chainId_]) != address(0)) {
            (, int256 value, , , ) = gasPriceOracle[chainId_].latestRoundData();
            return uint256(value);
        }

        return gasPrice[chainId_];
    }
```

It's quite possible that the oracle data is stale as we've seen in Terra crash. We can't just use the price without checking `updatedAt>0` or `answer>0`.

Impact Using stale prices for calculating fees can result in overcharging or undercharging users.

Recommendation: Add checks for `answer` and `updatedAt`.

```
    (
      uint80 roundID,
      int256 value,
      ,
      uint256 updatedAt,

    ) = gasPriceOracle[chainId_].latestRoundData();

    require(value > 0, "Chainlink Malfunction");
    require(updatedAt != 0, "Incomplete round");
```

Superform:

Hans:

7.2.10 No bridge validator for 1inch or 0x swaps

Severity: Medium

Context: ERC4626FormImplementation.sol#L152

Description: Currently, there are no specific validators for 1inch or 0x swaps. If the SocketValidator is intended to be used with those types of operations, it won't work.

Socket: ZeroX Swap Implementation `https://etherscan.io/address/0x33BE2a7CF4Bb94d28131116F840d313Cab1eD2DA#writeContract`

Socket: 1Inch Swap Implementation 2 `https://etherscan.io/address/0x2ddf16BA6d0180e5357d5e170eF1917a01b41fc0`

These two implementations have operations `performDirectAction, performAction` and they can't be verified in SocketValidator.

Impact These swaps may not be functional.

Recommendation: Make it clear what `bridgeId` means, and check if 1inch or 0x can be of separate bridge id.

Superform:

Hans:

7.2.11 Possible reentrancy in `processPayload()`

Severity: Medium

Context: CoreStateRegistry.sol#L215

TwoStepsFormStateRegistry.sol#L182

Description: `CoreStateRegistry.processPayload()` doesn't conform to the CEI pattern as it updates the `payloadTracking` after processing deposit/withdrawal. As a result, a reentrancy attack would be possible with the compromised processor role and underlying token with a hook.

Impact The attackers might withdraw more funds by calling `processPayload()` several times.

Recommendation: `payloadTracking[payloadId_]` should be updated first before processing any deposits/withdrawals.

Superform:

Hans:

7.2.12 Unsafe use of `transfer()` with `IERC20`

Severity: Medium

Context: CoreStateRegistry.sol#L483 CoreStateRegistry.sol#L590

Description: In `CoreStateRegistry.sol`, it uses `IERC20.transfer()` to deposit underlying tokens.

```
File: src\crosschain-data\extensions\CoreStateRegistry.sol
482:            if (underlying.balanceOf(address(this)) >= multiVaultData.amounts[i]) {
483:                underlying.transfer(superforms[i], multiVaultData.amounts[i]); //@audit unsafe

589:        if (underlying.balanceOf(address(this)) >= singleVaultData.amount) {
590:            underlying.transfer(superform_, singleVaultData.amount); //@audit unsafe
```

As we can see from here, some tokens don't revert on failure, but return `false` instead.

Impact A failed transfer might be considered successful and affect the protocol seriously.

Recommendation: Use OpenZeppelin's `SafeERC20` library.

Superform:

Hans:

7.2.13 `revokeProtocolAdminRole()` might remove `PROTOCOL_ADMIN_ROLE` permanently

Severity: Medium

Context: SuperRBAC.sol#L79

Description: `revokeProtocolAdminRole()` is used to remove the protocol admin role.

```
    function revokeProtocolAdminRole(address admin_) external override {
        revokeRole(PROTOCOL_ADMIN_ROLE, admin_);
    }
```

If this function is called when there is only one `PROTOCOL_ADMIN_ROLE`, the protocol won't have that role forever.

Impact It would be impossible to perform any admin-specific functionalities forever.

Recommendation: Revise the `revokeProtocolAdminRole()` to ensure at least one one protocol admin remains in the system.

Superform:

Hans:

7.2.14 Admin-level vulnerabilities

Severity: Medium

Context: CoreStateRegistry.sol#L151

Description: In the current protocol, all the cross-chain interactions are done by the processor role through `processPayload()`. As a result, user funds might be stuck by malicious processors and there are more scenarios that might affect the funds by compromised roles.

Although we assume the admin is trusted, users should acknowledge it before interacting with the protocol.

Impact The admin can change the protocol's behavior in unexpected ways.

Recommendation: The centralization risk exists in most protocols and the protocol documentation should include a clear explanation about that.

Superform:

Hans:

7.3 Low Issues

7.3.1 Check array length mismatch

```
File: src\SuperformFactory.sol
127:         superformIds_ = new uint256[](formBeaconIds_.length);

File: src\settings\SuperRegistry.sol
220:         for (uint256 i = 0; i < bridgeId_.length; i++) {
```

7.3.2 Should emit logs for important config updates

```
File: src\payments\PaymentHelper.sol
108:     function updateChainConfig(

File: src\crosschain-data\adapters\layerzero\LayerzeroImplementation.sol
254:             lzEndpoint = ILayerZeroEndpoint(endpoint_);
```

7.3.3 Inconsistency between the comment and implementation

The comment says it will return 0 if no oracle is set, but the implementation returns the preset value instead.

```
File: src\payments\PaymentHelper.sol
667:     /// @dev returns `0` - if no oracle is set
```

7.3.4 Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

```
File: BaseForm.sol
61:     function initialize(address superRegistry_, address vault_) external initializer {
```

7.3.5 Unsafe approve()

```
File: crosschain-liquidity/MultiTxProcessor.sol
74:             IERC20(approvalToken_).approve(to, amount_);

File: forms/ERC4626FormImplementation.sol
181:         IERC20(vars.collateral).approve(vault, singleVaultData_.amount);
242:         IERC20(v.asset()).approve(vaultLoc, singleVaultData_.amount);
```

7.4 Informational Findings

7.4.1 safeApprove() is deprecated

```
File: crosschain-liquidity/LiquidityHandler.sol
71:             token.safeApprove(bridge_, amount_);
```

7.4.2 Unused error

```
File: src\utils\Error.sol
31:     error NOT_FORM_STATE_REGISTRY();
```

7.5 Gas Optimizations

7.5.1 Using bools for storage incurs overhead

Use uint256(1) and uint256(2) for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

```
File: SuperPositions.sol
25:     bool public dynamicURIFrozen;

File: forms/FormBeacon.sol
24:     bool public paused;
```

7.5.2 Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

```
File: SuperformFactory.sol
127:        superformIds_ = new uint256[](formBeaconIds_.length);
128:        superforms_ = new address[](formBeaconIds_.length);
130:        for (uint256 i = 0; i < formBeaconIds_.length; ) {
203:        uint256 len = superformIds_.length;
226:        uint256 len = superformIds_.length;
236:        forms_ = formBeacons.length;
241:        superforms_ = superforms.length;

File: SuperformRouter.sol
82:        for (uint256 i; i < req.dstChainIds.length; ) {
105:        for (uint256 i = 0; i < req.dstChainIds.length; i++) {
156:        for (uint256 i; i < req.dstChainIds.length; ) {
178:        for (uint256 i = 0; i < req.dstChainIds.length; i++) {
253:        for (uint256 j = 0; j < req.superformsData.liqRequests.length; ) {
653:        uint256 len = vaultData_.superformIds.length;
738:        for (uint256 i; i < superforms.length; ) {
805:        uint256 len = superformsData_.amounts.length;
806:        uint256 liqRequestsLen = superformsData_.liqRequests.length;
814:            !(superformsData_.superformIds.length == superformsData_.amounts.length &&
815:                superformsData_.superformIds.length == superformsData_.maxSlippages.length)
850:        uint256 len = superformsData_.amounts.length;
851:        uint256 liqRequestsLen = superformsData_.liqRequests.length;
862:            !(superformsData_.superformIds.length == superformsData_.amounts.length &&
863:                superformsData_.superformIds.length == superformsData_.maxSlippages.length)

File: crosschain-data/BaseStateRegistry.sol
158:        for (uint8 i = 1; i < ambIds_.length; ) {

File: crosschain-data/extensions/CoreStateRegistry.sol
235:        uint256 l1 = superformIds.length;
414:        for (uint256 i; i < multiVaultData.superformIds.length; ) {
473:        uint256 numberOfVaults = multiVaultData.superformIds.length;

File: crosschain-liquidity/MultiTxProcessor.sol
92:        for (uint256 i; i < txData_.length; ) {

File: libraries/PayloadUpdaterLib.sol
30:        for (uint256 i; i < newAmount.length; ) {

File: payments/PaymentHelper.sol
194:        for (uint256 i; i < req_.dstChainIds.length; ) {
211:                totalDstGas += _estimateUpdateCost(req_.dstChainIds[i],
↪  req_.superformsData[i].superformIds.length);
217:                    req_.superformsData[i].superformIds.length
229:                req_.superformsData[i].superformIds.length
248:        for (uint256 i; i < req_.dstChainIds.length; ) {
309:        if (isDeposit) totalDstGas += _estimateUpdateCost(req_.dstChainId,
↪  req_.superformsData.superformIds.length);
313:        totalDstGas += _estimateDstExecutionCost(isDeposit, req_.dstChainId,
↪  req_.superformsData.superformIds.length);
316:        if (isDeposit) srcAmount += _estimateAckProcessingCost(1,
↪  req_.superformsData.superformIds.length);
386:        for (uint256 i; i < req_.superformData.superformIds.length; ) {
527:        for (uint256 i; i < liqReq_.length; ) {
553:        for (uint256 i; i < req_.length; ) {

File: settings/SuperRegistry.sol
220:        for (uint256 i = 0; i < bridgeId_.length; i++) {
234:        for (uint256 i; i < ambId_.length; i++) {
250:        for (uint256 i; i < registryId_.length; i++) {
```

7.5.3 Don't initialize variables with default value

```
File: SuperformFactory.sol
130:        for (uint256 i = 0; i < formBeaconIds_.length; ) {
206:        for (uint256 i = 0; i < len; i++) {
229:        for (uint256 i = 0; i < len; i++) {


File: SuperformRouter.sol
105:        for (uint256 i = 0; i < req.dstChainIds.length; i++) {
178:        for (uint256 i = 0; i < req.dstChainIds.length; i++) {
253:        for (uint256 j = 0; j < req.superformsData.liqRequests.length; ) {
822:        for (uint256 i = 0; i < len; ) {

File: settings/SuperRegistry.sol
220:        for (uint256 i = 0; i < bridgeId_.length; i++) {
```

7.5.4 `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 5 gas per loop

```
File: SuperformFactory.sol
206:        for (uint256 i = 0; i < len; i++) {
229:        for (uint256 i = 0; i < len; i++) {

File: SuperformRouter.sol
105:        for (uint256 i = 0; i < req.dstChainIds.length; i++) {
178:        for (uint256 i = 0; i < req.dstChainIds.length; i++) {

File: settings/SuperRegistry.sol
220:        for (uint256 i = 0; i < bridgeId_.length; i++) {
234:        for (uint256 i; i < ambId_.length; i++) {
250:        for (uint256 i; i < registryId_.length; i++) {
```

7.5.5 Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

```
File: SuperformRouter.sol
35:     uint8 public constant STATE_REGISTRY_TYPE = 1;

File: payments/PaymentHelper.sol
34:     uint32 public constant TIMELOCK_FORM_ID = 1;

File: settings/SuperRBAC.sol
15:     uint8 public constant STATE_REGISTRY_TYPE = 2;
16:     bytes32 public constant SYNC_REVOKE = keccak256("SYNC_REVOKE");
19:     bytes32 public constant override EMERGENCY_ADMIN_ROLE = keccak256("EMERGENCY_ADMIN_ROLE");
20:     bytes32 public constant override PAYMENT_ADMIN_ROLE = keccak256("PAYMENT_ADMIN_ROLE");
21:     bytes32 public constant override SWAPPER_ROLE = keccak256("SWAPPER_ROLE");
22:     bytes32 public constant override CORE_CONTRACTS_ROLE = keccak256("CORE_CONTRACTS_ROLE");
23:     bytes32 public constant override PROCESSOR_ROLE = keccak256("PROCESSOR_ROLE");
24:     bytes32 public constant override TWOSTEPS_PROCESSOR_ROLE = keccak256("TWOSTEPS_PROCESSOR_ROLE");
25:     bytes32 public constant override UPDATER_ROLE = keccak256("UPDATER_ROLE");
26:     bytes32 public constant override MINTER_ROLE = keccak256("MINTER_ROLE");
27:     bytes32 public constant override BURNER_ROLE = keccak256("BURNER_ROLE");
28:     bytes32 public constant override MINTER_STATE_REGISTRY = keccak256("MINTER_STATE_REGISTRY");

File: settings/SuperRegistry.sol
32:     bytes32 public constant override SUPER_ROUTER = keccak256("SUPER_ROUTER");
33:     bytes32 public constant override SUPERFORM_FACTORY = keccak256("SUPERFORM_FACTORY");
34:     bytes32 public constant override PAYMASTER = keccak256("PAYMASTER");
35:     bytes32 public constant override PAYMENT_HELPER = keccak256("PAYMENT_HELPER");
36:     bytes32 public constant override CORE_STATE_REGISTRY = keccak256("CORE_STATE_REGISTRY");
37:     bytes32 public constant override TWO_STEPS_FORM_STATE_REGISTRY =
↪   keccak256("TWO_STEPS_FORM_STATE_REGISTRY");
38:     bytes32 public constant override FACTORY_STATE_REGISTRY = keccak256("FACTORY_STATE_REGISTRY");
39:     bytes32 public constant override ROLES_STATE_REGISTRY = keccak256("ROLES_STATE_REGISTRY");
40:     bytes32 public constant override SUPER_POSITIONS = keccak256("SUPER_POSITIONS");
41:     bytes32 public constant override SUPER_RBAC = keccak256("SUPER_RBAC");
42:     bytes32 public constant override MULTI_TX_PROCESSOR = keccak256("MULTI_TX_PROCESSOR");
43:     bytes32 public constant override TX_PROCESSOR = keccak256("TX_PROCESSOR");
44:     bytes32 public constant override TX_UPDATER = keccak256("TX_UPDATER");
```