SPARK DIAMOND PAU V1.11 AUDIT

03-03-2026 - 13-03-2026

Table of contents

1. Project brief

Title	Description
Client	Spark
Project name	Spark Diamond PAU v1.11 Audit
Timeline	03-03-2026 - 13-03-2026

Project Log

diamond-pau

Date	Commit Hash	Note
04-03-2026	6b0958f575d6fc99311df5326353f3202fba95b5	**audited commit (release v1.11.0-beta.1)**
13-03-2026	697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e	**fix commit (production release v1.11.0)**

v1.10.0

Date	Commit Hash	Note
03-03-2026	984ec546fb2c98ed729ae91d2d73e97dedbf111f	**reference release (v1.10.0) used for differential comparison**

Short Overview

Together with Spark, Unvariant performed a focused security review of stage 1 of Spark's migration to a diamond architecture from 03-03-2026 to 13-03-2026.

In this stage, integrations and related logic were extracted from `MainnetController` and `ForeignController` and moved into dedicated libraries. The main objective of the review was to confirm that this refactor preserved behavior and did not introduce unintended logic changes or functional gaps.

This report covers the issues identified during that process and the remediation status confirmed with the Spark team.

Project Scope

- ALMProxy.sol
- ALMProxyFreezable.sol
- ForeignController.sol
- MainnetController.sol
- OTCBuffer.sol
- RateLimitHelpers.sol
- RateLimits.sol
- WEETHModule.sol
- AaveLib.sol
- ApproveLib.sol
- CCTPLib.sol
- CentrifugeLib.sol
- CurveLib.sol
- DAIUSDSLib.sol
- ERC4626Lib.sol
- ERC7540Lib.sol
- FarmLib.sol
- LayerZeroLib.sol
- MapleLib.sol
- MerklLib.sol
- OTCLib.sol
- PSM3Lib.sol
- PSMLib.sol
- PendleLib.sol
- SparkVaultLib.sol
- SuperstateLib.sol
- TransferAssetLib.sol
- USDELib.sol
- USDSLib.sol
- UniswapV3Lib.sol
- UniswapV4Lib.sol
- WEETHLib.sol
- WSTETHLib.sol
- WrapProxyETHLib.sol
- UniswapV3OracleLib.sol
- UniswapV3UtilsLib.sol

2. Finding severity breakdown

All vulnerabilities identified in the audit are grouped by their potential impact using the following severity scale:

Severity	Description
Critical	Vulnerabilities that enable theft of funds, lock user access to assets, or otherwise cause funds to be moved away from their rightful owners.
High	Bugs that can halt contract operation and typically require manual state intervention or a contract replacement to recover.
Medium	Bugs that disrupt intended logic or open DoS vectors but do not directly lead to loss of funds.
Low	Low-impact concerns with no immediate material effect that are straightforward to correct.
Informational	Best-practice recommendations and quality improvements with negligible security impact.

After the Client reviews the findings reported by Unvariant, each item receives one of the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Medium	3 (2 fixed, 1 acknowledged)
Low	11 (5 fixed, 6 acknowledged)
Informational	3 (0 fixed, 3 acknowledged)
Total	17 (7 fixed, 10 acknowledged)

4. Conclusion

A total of 17 findings were identified during this review. Each finding was reviewed by the Spark team and either acknowledged with supporting comments (for example, planned fixes in a subsequent release or rationale for current behavior) or remediated. Unvariant team also carefully audited new integrations not presented in Spark ALM controller to ensure that the entire release is safe.

Remediations were implemented in commit [697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e]. Unvariant reviewed these changes and did not identify new issues introduced by the fixes.

Unvariant considers this release to be safe and ready for deployment.

5. Findings report

MEDIUM-01	**UniswapV3Lib.removeLiquidity() can be DOSed by fee accrual**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

Lines:

- UniswapV3Lib.sol#L418
- UniswapV3Lib.sol#L846
- NonfungiblePositionManager.sol#L275
- UniswapV3Lib.sol#L420
- UniswapV3Lib.sol#L872
- NonfungiblePositionManager.sol#L354
- UniswapV3Lib.sol#L888

The `UniswapV3Lib.removeLiquidity()` function performs three steps in sequence: `NonfungiblePositionManager.decreaseLiquidity()` (burns principal), `NonfungiblePositionManager.collect()` (withdraws principal + all accrued fees), then a slippage check on proxy balance deltas.

The problem is that `UniswapV3Lib._checkSlippage()` operates on `endingBalance - startingBalance`, which is the total amount received by the proxy: principal + accrued fees:

```
function _checkSlippage(
        uint256 maxSlippage,
        uint256 startingBalance,
        uint256 endingBalance,
        uint256 minAmount
) internal pure {
        require(
                minAmount >= ((endingBalance - startingBalance) * maxSlippage) / 1e18,
                //              ^-- principal + fees
                "UniswapV3Lib/min-amount-below-bound"
        );
}
```

Meanwhile, `min.amount0` / `min.amount1` are also passed into `NonfungiblePositionManager.decreaseLiquidity()`, where Uniswap validates them against principal only:

```
function decreaseLiquidity(DecreaseLiquidityParams calldata params)
        // ...
{
        // ...

        (amount0, amount1) = pool.burn(position.tickLower, position.tickUpper,
params.liquidity);

        require(amount0 >= params.amount0Min && amount1 >= params.amount1Min, 'Price
slippage check');
```

After `NonfungiblePositionManager.decreaseLiquidity()`, the `NonfungiblePositionManager` stores both principal and accrued fees into `tokensOwed` but transfers nothing. The `NonfungiblePositionManager.collect()` function then sends the full `tokensOwed` (principal + fees) to the proxy because the library passes `type(uint128).max`:

```
function _callCollect(address proxy, address positionManager, uint256 tokenId)
        internal
        returns (TokenAmounts memory amounts)
{
        bytes memory result = IALMProxy(proxy).doCall(
                positionManager,
                abi.encodeCall(
                        INonfungiblePositionManager.collect,
                        INonfungiblePositionManager.CollectParams({
                                tokenId    : tokenId,
                                recipient  : proxy,
                                amount0Max : type(uint128).max,
                                amount1Max : type(uint128).max
                                // ^-- principal + fees
                        })
                )
        );

        // ...

}
```

This creates two conflicting constraints on the relayer's `min` values:

- `NonfungiblePositionManager`: `min.amount0 ≤ principal`
- `UniswapV3Lib`: `min.amount0 ≥ (principal + fees) × maxSlippage / 1e18`

For both to hold: `principal ≥ (principal + fees) × maxSlippage / 1e18`.

For example with `maxSlippage = 0.95e18`, principal = 1000 USDC and accrued fees = 100 USDC:

- Uniswap requires: `min ≤ 1000`
- Library requires: `min ≥ 1100 × 0.95 = 1045`

slippage that's ~5.3% fees-to-principal. Once a position's accrued fees exceed a breakage fraction of its principal (determined by `maxSlippage`), `UniswapV3Lib.removeLiquidity()` becomes permanently uncallable for relayers.

Additionally, the withdraw rate limit is decreased by `collected.amount0/1`, the combined principal + fees amount. This means fee income consumes the withdrawal quota. During high-fee periods a position that has barely moved in principal can consume a disproportionate share of the rate limit, blocking other legitimate withdrawals.

Recommendation

We recommend decoding the return values of `decreaseLiquidity` (the principal amounts) and use those for the slippage check instead of post-collect balance deltas or use different approach. We also recommend excluding fees from the decrease amount for withdrawals limit.

MEDIUM-02	**The `manager()` getter was removed from Centrifuge vaults**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

Lines:

- CentrifugeLib.sol#L31
- CentrifugeLib.sol#L144

The `CentrifugeLib.transferShares()` resolves the Spoke address at runtime by calling `ICentrifugeV3VaultLike.manager()` on the vault, then `spoke()` on the returned manager. The interface declares `manager()` as:

```
interface ICentrifugeV3VaultLike {
        // ...
    function manager() external view returns (address);
    // ...
}
```

Centrifuge renamed this field from `manager` to `baseManager` and created special getter for `manager()` in PR #409. However, Centrifuge removed the `manager()` getter in PR #563. The change is already live - for example the deCRDX vault on Optimism (v3.1.0) no longer has the `manager()` getter. The auto-generated getter is now `baseManager()`.

Calling `manager()` on a vault deployed with the updated Centrifuge code reverts because there is no `manager()` function. This breaks `CentrifugeLib.transferShares()` and, by extension, `MainnetController.transferSharesCentrifuge()` and `ForeignController.transferSharesCentrifuge()`.

Existing fork tests pass only because they run against older vault deployments that still have the original `manager` field. Any new vault integration using updated Centrifuge contracts will fail at runtime.

Recommendation

We recommend updating the interface to match the current Centrifuge vault ABI:

```
- function manager() external view returns (address);
+ function baseManager() external view returns (address);
```

MEDIUM-03	**Uniswap V3 removeLiquidity can be executed at manipulated prices**	Acknowledged

Description

Lines:

- UniswapV3Lib.sol#L424-L436
- UniswapV3Lib.sol#L888
- MainnetController.sol#L657

The `UniswapV3Lib.removeLiquidity()` only checks that the relayer-supplied `min` values are close to the amounts actually collected. This makes the guard self-referential: the relayer chooses `min`, and the contract compares `min` to the post-execution outcome rather than to any independent price reference such as the pool's TWAP oracle.

Current protection is at the end of `UniswapV3Lib.removeLiquidity()`:

```
function removeLiquidity(...) external returns (TokenAmounts memory amounts) {
        ...

        _checkSlippage(
                maxSlippages[pool],
                startingBalances.amount0,
                endingBalances.amount0,
                min.amount0
        );

        _checkSlippage(
                maxSlippages[pool],
                startingBalances.amount1,
                endingBalances.amount1,
                min.amount1
        );
        ...
}

...

...

function _checkSlippage(
        uint256 maxSlippage,
        uint256 startingBalance,
        uint256 endingBalance,
        uint256 minAmount
) internal pure {
        require(
                minAmount >= ((endingBalance - startingBalance) * maxSlippage) / 1e18,
                "UniswapV3Lib/min-amount-below-bound"
        );
}
```

the current `decreaseLiquidity()` burn and any previously accumulated position fees. Uniswap V3's `decreaseLiquidity()` internally enforces that burn amounts are `>= min.amount0` (same for token 1), and the subsequent `UniswapV3Lib._collectAll()` gives at least those burn amounts. Combining these guarantees with the contract's check:

```
collectedDelta0 >= decreaseAmount0 >= min.amount0 >= collectedDelta0 * maxSlippage / 1e18
```

The outer inequality `collectedDelta0 >= collectedDelta0 * maxSlippage / 1e18` holds whenever `maxSlippage <= 1e18`, which is always enforced by the contract. The relayer can set `min` to any value in the range `[collectedDelta * maxSlippage / 1e18, collectedDelta]` and all checks pass, regardless of how far the pool's spot price has deviated from fair value.

For comparison, the other Uniswap V3 operations in this library have independent, TWAP-anchored protections:

- `UniswapV3Lib.swap()` computes the swap price limit as `twapTick +/- tickDelta`, where `tickDelta` is relayer-supplied but bounded by the governance-set `poolParams.swapMaxTickDelta`.
- `UniswapV3Lib.addLiquidity()` validates the relayer-supplied `min` amounts against TWAP-derived expected amounts in `UniswapV3Lib._validateAddLiquidityMinAmounts()`.

This means removals can execute at any spot price, including prices temporarily shifted by a sandwich attacker. This allows value extraction from the protocol's Uniswap V3 positions through a classic sandwich around liquidity removal.

Recommendation

We recommend validating the relayer-supplied min amounts against TWAP-derived expected withdrawal amounts, similar to `UniswapV3Lib._getExpectedAmounts()` approach in `UniswapV3Lib.addLiquidity()`.

Customer's Response

Acknowledged. The Grove team wants to prioritize removing liquidity quickly and without additional barriers over limits that may result in DOS vectors. In the UniswapV4 integration, slippage is limited by the admin's choice of a max tick width. A future review of the UniswapV3 integration may either adopt that pattern, or consider the tradeoffs between liquidity removal barriers and relayer-controlled slippage.

LOW-01	**Hardcoded `MAX_FEE = 0` may break CCTP transfers**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

Lines:

- CCTPLib.sol#L61
- CCTPLib.sol#L151
- TokenMessengerV2.sol#L347

The `CCTPLib.sol` library defines `MAX_FEE` as a constant set to zero:

```
bytes32 public constant DESTINATION_CALLER     = 0;      // 0 means anyone can relay
uint256 public constant MAX_FEE                = 0;      // 0 for standard burns (no fast
burn fee)
uint32  public constant MAX_FINALITY_THRESHOLD = 2_000;  // 2_000 for standard
(finalized) messages
```

This value is passed directly into `TokenMessengerV2.depositForBurn()` every time a cross-chain USDC transfer is initiated:

```
function transferUSDCToCCTP(
        address _tokenMessenger,
        uint256 _amount,
        uint32  _destinationDomain,
        bytes32 _recipient
) external returns (bytes32) {

        // ...

        // Call depositForBurn on TokenMessenger
        return ITokenMessenger(_tokenMessenger).depositForBurn(
                _amount,
                _destinationDomain,
                _recipient,
                USDC,
                MAX_FEE, // <-- zero fee
                MAX_FINALITY_THRESHOLD
        );
}
```

`TokenMessengerV2._depositForBurn()` validates `maxFee` against a governance-controlled `minFee` rate. This functionality was introduced in PR #68.

```
require(_amount > 0, "Amount must be nonzero");
require(_mintRecipient != bytes32(0), "Mint recipient must be nonzero");
require(_maxFee < _amount, "Max fee must be less than amount");

if (minFee > 0) {
        // ^-- minFee can be non-zero
    require(
        _maxFee >= _calcMinFeeAmount(_amount),
        "Insufficient max fee"
    );
}
```

If `minFee` becomes non-zero, the `TokenMessengerV2._calcMinFeeAmount()` function calculates the proportional minimum fee amount and limits it to a value of at least 1:

```
function _calcMinFeeAmount(
    uint256 _amount
) internal view returns (uint256) {
    uint256 _minFeeAmount = _amount.mul(minFee) / MIN_FEE_MULTIPLIER;
    return _minFeeAmount == 0 ? 1 : _minFeeAmount;
    // ^-- at least 1
}
```

The `minFee` functionality is already live in some of production `TokenMessengerV2` deployments, including the Plume implementation. On any chain where `minFee > 0`, every CCTP transfer originating from both `MainnetController` and `ForeignController` would revert because `MAX_FEE` is hardcoded to zero and can never satisfy the required minimum fee. Fixing this would require redeploying library.

Recommendation

We recommend replacing the hardcoded constant with a configurable value that admins can update.

LOW-02	**Missing validation for zero `burnLimit` in `CCTPLib.transfer()`**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

Line: CCTPLib.sol#L103

`CCTPLib.transfer()` fetches the per-message burn limit from CCTP's `TokenMinter` and uses it to chunk large transfers:

```
uint256 burnLimit =
ICCTPTokenMinterLike(ICCTPLike(cctp).localMinter()).burnLimitsPerMessage(usdc);

while (usdcAmount > 0) {
    uint256 amount = usdcAmount > burnLimit ? burnLimit : usdcAmount;

    _initiateTransfer(proxy, cctp, usdc, amount, recipient, destinationDomain);

    usdcAmount -= amount;
}
```

Circle can set `burnLimitsPerMessage` to zero via `TokenController.setMaxBurnAmountPerMessage()`. If `burnLimit` is zero, the call reaches `TokenMessengerV2.depositForBurn()` with `amount = 0`, which reverts with:

```
require(_amount > 0, "Amount must be nonzero");
```

The error `"Amount must be nonzero"` gives no indication that the real problem is a zero burn limit from the CCTP minter.

Recommendation

We recommend addressing this or documenting the case.

LOW-03	**Missing `maxSlippage` check in `UniswapV3Lib.removeLiquidity()` disables slippage protection**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

The `UniswapV3Lib.addLiquidity()` validates that `maxSlippage` is configured before proceeding:

```
        uint256 maxSlippage = maxSlippages[pool];

        require(target.amount0 > 0 || target.amount1 > 0, "UniswapV3Lib/zero-amount");
        require(maxSlippage != 0,                         "UniswapV3Lib/max-slippage-not-
set");
```

However, the `UniswapV3Lib.removeLiquidity()` has no such check. It passes `maxSlippages[pool]` straight into `UniswapV3Lib._checkSlippage()`. When `maxSlippage == 0` (pool not configured or accidentally zeroed), `UniswapV3Lib._checkSlippage()` reduces to a no-op:

```
function _checkSlippage(
        uint256 maxSlippage,
        uint256 startingBalance,
        uint256 endingBalance,
        uint256 minAmount
) internal pure {
        require(
                minAmount >= ((endingBalance - startingBalance) * maxSlippage) / 1e18,
                "UniswapV3Lib/min-amount-below-bound"
        );
        // ^-- equivalent to: require(minAmount >= 0) — always true
}
```

With `maxSlippage = 0` the right side is always `0`, so `require(minAmount >= 0)` passes for any input. A relayer can call `MainnetController.removeLiquidityUniswapV3()` with `min.amount0 = 0, min.amount1 = 0` and the library won't enforce any slippage bound.

If `maxSlippages[pool]` is zero (not yet configured, or reset to zero), liquidity removals have no protocol-level slippage protection. A relayer can remove liquidity with zero minimums, exposing the position to sandwich attacks.

Recommendation

We recommend adding `maxSlippage != 0` check into the `UniswapV3Lib._validateRemoveLiquidityParams()` function.

LOW-04	**Cancel and claim paths blocked when rate-limit `maxAmount` is zero**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

Lines:

- CentrifugeLib.sol#L84
- CentrifugeLib.sol#L94
- CentrifugeLib.sol#L106
- CentrifugeLib.sol#L116
- ERC7540Lib.sol#L69
- ERC7540Lib.sol#L94

All cancel and claim functions in `CentrifugeLib` and `ERC7540Lib` gate execution with `_rateLimitExists()`:

```
function _rateLimitExists(IRateLimits rateLimits, bytes32 key) internal view {
        require(
                rateLimits.getRateLimitData(key).maxAmount > 0,
                "CentrifugeLib/invalid-action"
        );
        // ^-- checks that the limit exists
}
```

This check is used in `CentrifugeLib.cancelDepositRequest()`, `CentrifugeLib.claimCancelDepositRequest()`, `CentrifugeLib.cancelRedeemRequest()`, `CentrifugeLib.claimCancelRedeemRequest()`, `ERC7540Lib.claimDeposit()`, `ERC7540Lib.claimRedeem()` functions.

The problem is that `_rateLimitExists()` ties "is this action allowed" to "is the rate limit quota non-zero". Governance might set `maxAmount = 0` to pause new deposits/redeems, but this also blocks cancellations and claims for already-pending requests.

Scenario:

1. Relayer calls `MainnetController.requestDepositERC7540(token, 1M)`. Assets go into Centrifuge's pending queue, rate limit decreases.
2. Governance sets the deposit rate limit `maxAmount` to 0 to block new deposits.
3. Relayer tries to cancel the pending request: `CentrifugeLib.cancelDepositRequest()` reverts with `"CentrifugeLib/invalid-action"` because `maxAmount == 0`.
4. Assets are stuck in the pending request with no way to unwind through the controller.

Impact

Setting `maxAmount = 0` as an emergency measure blocks not just new requests but also the ability to cancel or claim-cancel already-pending ones. Pending capital becomes stuck until governance re-enables a non-zero limit.

Recommendation

We recommend addressing this or documenting the case.

LOW-05	**Small `TokenMinter.burnLimitsPerMessage()` makes `CCTPLib.transfer()` gas-intense**	Fixed at 697a8dbb2cd5b46ae7b5c618e1d9c7efb112a83e

Description

Lines:

- CCTPLib.sol#L103
- CCTPLib.sol#L107

`CCTPLib.transfer()` splits USDC transfers into chunks based on `TokenMinter.burnLimitsPerMessage()`:

```
        // If amount is larger than limit it must be split into multiple calls.
        uint256 burnLimit =
ICCTPTokenMinterLike(ICCTPLike(cctp).localMinter()).burnLimitsPerMessage(usdc);

        while (usdcAmount > 0) {
                uint256 amount = usdcAmount > burnLimit ? burnLimit : usdcAmount;
                // ^-- the number of loop iterations directly depends on the burnLimit

                _initiateTransfer(
                        proxy,
                        cctp,
                        usdc,
                        amount,
                        recipient,
                        destinationDomain
                );

                usdcAmount -= amount;
        }
```

The loop runs `ceil(usdcAmount / burnLimit)` times. Each iteration does a full external call through the `ALMProxy` into CCTP's `TokenMessengerV2.depositForBurn()`. Circle's `tokenController` role can change `burnLimitsPerMessage` at any time via `TokenController.setMaxBurnAmountPerMessage()`.

ALM rate limits allow multi-million USDC transfers. If Circle reduces `burnLimitsPerMessage` to a small value (e.g. from current 10M to 1K USDC), a 10M USDC transfer goes from 1 iteration to 10000, easily exceeding the block gas limit. The transaction just reverts with an out-of-gas error, no clear indication of the root cause.

Recommendation

We recommend addressing this or documenting the case.

LOW-06	**Rate limit exhaustion in Ethena integration**	Acknowledged

Description

Lines:

- USDELib.sol#L65
- USDELib.sol#L79

The Ethena integration includes setting up token approvals for minting and burning operations. These actions are executed in separate transactions by designated Ethena roles. The functions `USDELib.prepareMint()` and `USDELib.prepareBurn()` approve token amounts to the Ethena minter in preparation for these operations.

However, calling these functions consumes the contract's rate limit without performing the actual mint or burn. Since approvals can be overwritten by subsequent calls, this can lead to unnecessary rate limit consumption and potentially exhausting the limit without any minting or burning taking place.

This creates a DoS risk: a relayer could repeatedly call the prepare functions, draining the rate limit and blocking minting and burning operations.

Recommendation

We recommend canceling any existing non-zero allowance before applying new limits:

```
 function prepareMint(
         // ...
 ) external {
-        IRateLimits(rateLimits).triggerRateLimitDecrease(LIMIT_USDE_MINT, usdcAmount);
+        uint256 prevAllowance = IERC20Like(usdc).allowance(proxy, minter);
+        if (prevAllowance > 0) {
+            IRateLimits(rateLimits).triggerRateLimitIncrease(LIMIT_USDE_MINT,
prevAllowance);
+        }
+        IRateLimits(rateLimits).triggerRateLimitDecrease(LIMIT_USDE_MINT, usdcAmount);
         _approve(address(usdc), address(ethenaMinter), usdcAmount);
 }
```

The same fix can be applied to the `prepareBurn()` function.

Customer's Response

Acknowledged. A relayer that DOSes the system can be removed with the removeRelayer function.

LOW-07	**Excess ETH refund from Centrifuge cross-chain transfer goes to ALMProxy, not relayer**	Acknowledged

Description

Lines:

- CentrifugeLib.sol#L147
- ALMProxy.sol#L47
- Spoke.sol#L117
- ISpoke.sol#L188

When a relayer calls `MainnetController.transferSharesCentrifuge()`, the full `msg.value` is forwarded through the `ALMProxy` to the Centrifuge Spoke contract to pay for cross-chain messaging gas. Inside `CentrifugeLib.transferShares()`, the library calls the 6-parameter `crosschainTransferShares` function on the `Spoke` contract, routing through the `ALMProxy` via `doCallWithValue`:

```
IALMProxy(proxy).doCallWithValue{value: msg.value}(
        //              relayers msg.value --^
        spoke,
        abi.encodeCall(
                ISpokeLike.crosschainTransferShares,
                (
                        centrifugeId,
                        ICentrifugeV3VaultLike(token).poolId(),
                        ICentrifugeV3VaultLike(token).scId(),
                        recipient,
                        amount,
                        0
                )
        ),
        msg.value
);
```

The `doCallWithValue` executes the call from the `ALMProxy`'s context:

```
function doCallWithValue(address target, bytes memory data, uint256 value)
        external
        payable
        override
        onlyRole(CONTROLLER)
        returns (bytes memory result)
{
        result = target.functionCallWithValue(data, value);
}
```

Because the call goes from the `ALMProxy`, `msg.sender` inside the Spoke is the `ALMProxy` address. The 6-parameter `crosschainTransferShares` function on the `Spoke` contract internally sets

chain messaging layer refunds is therefore sent back to the `ALMProxy`, not to the relayer who originally attached the `ETH`.

In addition, the 6-parameter `crosschainTransferShares()` function is marked as legacy by the developers and it is worth using the new implementation of this function.

Recommendation

We recommend using the `Spoke`'s 8-parameter `crosschainTransferShares()` function that accepts an explicit `refund` address.

Customer's Response

Acknowledged. The recommendation will likely be adopted in a subsequent update to the integration lib/facet.

LOW-08	**Rate limit capacity is not restored on Centrifuge deposit/redeem cancellation**	Acknowledged

Description

Lines:

- CentrifugeLib.sol#L84
- CentrifugeLib.sol#L94
- CentrifugeLib.sol#L106
- CentrifugeLib.sol#L116

The `ERC7540Lib.requestDeposit()` function calls `triggerRateLimitDecrease` to consume rate limit capacity for the deposited amount. If the request is later cancelled via `CentrifugeLib.cancelDepositRequest()` followed by `CentrifugeLib.claimCancelDepositRequest()`, assets are returned to the `ALMProxy` but the rate limit is never restored. The cancel/claim paths only check `_rateLimitExists` and skip any call to `triggerRateLimitIncrease`.

The redeem flow has the same gap: `ERC7540Lib.requestRedeem()` decreases capacity, but `CentrifugeLib.cancelRedeemRequest()` + `CentrifugeLib.claimCancelRedeemRequest()` leave it consumed.

This is not exploitable by a relayer to extract value, but it can artificially lower limits and require admin intervention to manually reset rate limits after cancellations.

Recommendation

We recommend calling `triggerRateLimitIncrease` on the corresponding key inside `claimCancelDepositRequest` / `claimCancelRedeemRequest`, crediting back the claimed amount.

Customer's Response

Acknowledged. The recommendation will likely be adopted in a subsequent update to the integration lib/facet.

LOW-09	**Uniswap V3 swap uses legacy router interface without deadline protection**	Acknowledged

Description

Lines:

- UniswapV3Lib.sol#L25-L33

The `UniswapV3Lib` library defines the `ISwapRouter` interface whose `ExactInputSingleParams` struct omits the `deadline` field:

```
interface ISwapRouter {

    struct ExactInputSingleParams {
        address tokenIn;
        address tokenOut;
        uint24  fee;
        address recipient;
        // @audit no deadline param here
        uint256 amountIn;
        uint256 amountOutMinimum;
        uint160 sqrtPriceLimitX96;
    }

    function exactInputSingle(ExactInputSingleParams calldata params)
        external
        payable
        returns (uint256 amountOut);

}
```

This interface corresponds to the SwapRouter02 from the archived `Uniswap/swap-router-contracts` repository. The `SwapRouter02` intentionally removed the `deadline` field from the `ExactInputSingleParams` struct, expecting callers to enforce deadlines by wrapping calls in `multicall(uint256 deadline, bytes[] calldata data)`. However, `UniswapV3Lib._getSwapCallData()` encodes a direct call to `exactInputSingle` without any multicall wrapper.

As a result, swap transactions have no deadline enforcement. This is in contrast to the liquidity operations in the same library (`UniswapV3Lib.mint()`, `UniswapV3Lib.increaseLiquidity()`, `UniswapV3Lib.decreaseLiquidity()`), which all correctly accept and pass a `deadline` parameter via the `INonfungiblePositionManager` interface.

The canonical Uniswap V3 router defined in the v3-periphery repository includes `deadline` directly in its `ExactInputSingleParams`:

```
struct ExactInputSingleParams {
    address tokenIn;
    address tokenOut;
    uint24 fee;
    address recipient;
    uint256 deadline;
    uint256 amountIn;
    uint256 amountOutMinimum;
    uint160 sqrtPriceLimitX96;
}
```

Without a deadline, a swap transaction can sit in the mempool for an arbitrarily long period and then be executed when market conditions have changed in an unfavourable way.

Recommendation

We recommend switching to the canonical Uniswap V3 `SwapRouter` from the `v3-periphery` repository whose `ExactInputSingleParams` includes a `deadline` field.

Customer's Response

Acknowledged. The recommendation will likely be adopted in a subsequent update to the integration lib/facet.

LOW-10	**Rate limit capacity not restored on Maple redemption cancellation**	Acknowledged

Description

Line: MapleLib.sol#L55

When `MapleLib.requestRedemption()` is called, it decreases the `LIMIT_REDEEM` rate limit by the asset value of the shares being redeemed:

```
function requestRedemption(
        // ...
)
        external
{
        IRateLimits(rateLimits).triggerRateLimitDecrease(
                makeAddressKey(LIMIT_REDEEM, mapleToken),
                IMapleTokenLike(mapleToken).convertToAssets(shares)
        );
        // ^-- rate limit decreasing

        // ...

}
```

When that redemption is cancelled via `MapleLib.cancelRedemption()`, the function only checks that the rate limit key exists and it never calls `RateLimits.triggerRateLimitIncrease()` to restore the consumed capacity:

```
function cancelRedemption(
        // ...
)
        external
{
        require(
                IRateLimits(rateLimits).getRateLimitData(
                        makeAddressKey(LIMIT_REDEEM, mapleToken)
                ).maxAmount > 0,
                "MapleLib/invalid-action"
        );
        // ^-- only checks that the rate limit key exists

        // ...

}
```

The shares are returned to the `ALMProxy`, but the rate limit capacity consumed by the original `MapleLib.requestRedemption()` is gone. It only recovers via the natural `slope` over time.

isn't possible without storing it. A reasonable approach is to restore based on the current `convertToAssets` value, accepting the small discrepancy.

Recommendation

We recommend calling `RateLimits.triggerRateLimitIncrease()` after `MapleToken.removeShares()` succeeds, restoring the consumed capacity.

Customer's Response

Acknowledged. This is intended business functionality.

LOW-11	**`WSTETHLib.claimWithdrawal()` should use hint-based claim to avoid out-of-gas reverts**	Acknowledged

Description

Lines:

- WSTETHLib.sol#L106

The `WSTETHLib.claimWithdrawal()` calls Lido's `WithdrawalQueue.claimWithdrawal()` function, which accepts only one `requestId` argument. Lido's `claimWithdrawal()` internally calls `WithdrawalQueueBase._findCheckpointHint(_requestId, 1, getLastCheckpointIndex())`, which runs an on-chain binary search over all checkpoints to locate the correct finalization batch. It can lead to out-of-gas scenarios and Lido's own NatSpec warns about this:

```
/// ...
/// @dev use unbounded loop to find a hint, which can lead to OOG
// ...
function claimWithdrawal(uint256 _requestId) external {
        _claim(_requestId, _findCheckpointHint(_requestId, 1, getLastCheckpointIndex()),
msg.sender);
        _emitTransfer(msg.sender, address(0), _requestId);
}
```

However, Lido provides a hint-based alternative specifically designed to avoid this:

```
function claimWithdrawalsTo(
        uint256[] calldata _requestIds,
        uint256[] calldata _hints,
        // ^-- pre-computed hints
        address _recipient)
        external
{
        // ...
}
```

The hint can be computed off-chain for free by calling the view function `WithdrawalQueue.findCheckpointHints()`, then passed to `WithdrawalQueue.claimWithdrawalsTo()`. This skips the binary search entirely.

Recommendation

We recommend calling the `WithdrawalQueue.claimWithdrawalsTo()` with a pre-computed `hint` parameter from the relayer (computed off-chain via `WithdrawalQueue.findCheckpointHints()`).

Customer's Response

Acknowledged. Will address in a later release.

INFORMATIONAL-01	`UniswapV3Lib` tick bounds setters have non-obvious initialization ordering constraint	Acknowledged

Description

Lines:

- UniswapV3Lib.sol#L202
- UniswapV3Lib.sol#L220

The `UniswapV3Lib.setAddLiquidityLowerTickBound()` function validates `lowerTickBound` against the current upper bound:

```
function setAddLiquidityLowerTickBound(
        // ...
)
        external
{
        require(
                lowerTickBound >= MIN_TICK &&
                lowerTickBound < poolParams[pool].addLiquidityTickBounds.upper,
                "UniswapV3Lib/lower-tick-oob"
        );

        // ...
}
```

The `UniswapV3Lib.setAddLiquidityUpperTickBound()` validates `upperTickBound` against the current lower bound:

```
function setAddLiquidityUpperTickBound(
        // ...
)
        external
{
        require(
                upperTickBound <= MAX_TICK &&
                upperTickBound > poolParams[pool].addLiquidityTickBounds.lower,
                "UniswapV3Lib/upper-tick-oob"
        );

        // ...
}
```

When both tick bounds default to `0` (uninitialized pool), the initialization order matters. For positive tick ranges, the upper bound must be set first. For negative tick ranges, the lower bound must be set first. Setting them in the wrong order always reverts. This creates confusion during pool onboarding.

Recommendation

We recommend adding a function to set both tick bounds atomically in a single call (similar to `UniswapV4Lib.setTickLimits()`), or documenting the required initialization order clearly.

Customer's Response

Acknowledged, will address in a later release potentially.

INFORMATIONAL-02	**Duplicated key-generation helpers in `RateLimitHelpers`**	Acknowledged

Description

Line: RateLimitHelpers.sol#L28-L46

`RateLimitHelpers.sol` defines the same key-generation logic twice: once as file-level free functions and again as identical `internal pure` functions inside the `RateLimitHelpers` library.

All production code imports the free functions, while some tests and deployment scripts use the library. In addition, the library mirrors only 4 of the 6 free functions: `makeAddressUint16Key` and `makeAddressUint32Key` are missing. This duplication creates a risk that a future change applied to one set is not applied to the other, silently producing different keys depending on which import path is used.

Recommendation

We recommend removing the `RateLimitHelpers` library and updating the test and script files that reference it to import the free functions directly. If the library is intentionally retained to avoid a larger refactor in tests or scripts, this should be explicitly documented in comments.

Customer's Response

Acknowledged, will address in a later release potentially.

INFORMATIONAL-03	**Redundant `address()` cast on `proxy` parameter**	Acknowledged

Description

Lines:

- PSMLib.sol#L144
- WSTETHLib.sol#L102
- WSTETHLib.sol#L109

In `PSMLib`/`WSTETHLib` libraries, the `proxy` parameter is already declared as `address`, making the `address(proxy)` wrapping a no-op cast.

Recommendation

We recommend removing the redundant casts:

```
// PSMLib.sol
- abi.encodeCall(IPSMLike.sellGemNoFee, (address(proxy), usdcAmount))
+ abi.encodeCall(IPSMLike.sellGemNoFee, (proxy, usdcAmount))

// WSTETHLib.sol
- uint256 initialETHBalance = address(proxy).balance;
+ uint256 initialETHBalance = proxy.balance;

- IALMProxy(proxy).doCallWithValue(weth, "", address(proxy).balance - initialETHBalance);
+ IALMProxy(proxy).doCallWithValue(weth, "", proxy.balance - initialETHBalance);
```

Customer's Response

Acknowledged, will address in a later release.


UNJVARIANT