RENASCENCE

Phuture Finance Audit Report

Version 2.0

Audited by:

MiloTruck

HollaDieWaldfee

Windhustler

March 29, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Phuture Finance

Phuture V2 is a novel on-chain asset manager that builds on the first iteration of the protocol and introduces a number of significant new features and improvements:

- Multi-chain accounting - Index products have the ability to price and account for assets across any EVM enabled network through Proof of Asset (PoA) tokens.
- Multi-chain trading - Buying and selling batches of constituent assets can occur across any EVM enabled network through dex aggregators, ensuring absolute best execution.
- Contract-driven rebalancing - Multi-chain rebalancing is orchestrated from a homechain, where trades are generated and executed on each respective chain in a trustless manner.

2.2 Overview

Project	Phuture Finance
Repository	phuture-v2-contracts
Commit Hash	0ee3b0b909d0...
Mitigation Hash	e5de51538e77...
Date	12 February 2024 - 3 March 2024

2.3 Issues Found

Severity	**Count**
High Risk	11
Medium Risk	11
Low Risk	10
Informational	23
Total Issues	**55**

3 Findings Summary

ID	Description	Status
H-01	Indexing issues in `RebalancingLib` lead to undefined rebalancing behavior	Resolved
H-02	Rounding of sell amounts in `RebalancingLib` can make rebalancing stuck	Resolved
H-03	Missing validation of `orderCounts` in `RebalancingLib.previewRebalancingOrders()` can make rebalancing stuck	Resolved
H-04	Indexing issues in `ConfigBuilder` lead to corrupted anatomy data when rebalancing is finished	Resolved
H-05	`RedstoneChainlinkPriceOracle.valuation()` can return outdated valuation and funds can be stolen from the index	Resolved
H-06	`retryRedeem()` can be used to block the LayerZero pathway	Resolved
H-07	ETH balances of home and remote messenger can be drained	Resolved
H-08	Small sell orders cannot be executed and settled across chains	Resolved
H-09	`depositWithCommand()` undercounts `deposited` when `reserve` is ETH	Resolved
H-10	Reentrancy via callback from `SKIM` command leads to theft of `Vault` funds	Resolved
H-11	Stargate can not transfer asset amounts smaller than the conversion rate for the pool	Resolved
M-01	`send()` doesn't work on zkSync	Resolved
M-02	Tokens with non-string metadata revert	Resolved
M-03	`ConfigBuilder._prepareWithdrawals()` assumes that `reserve` is always sold and never bought	Resolved
M-04	`maxOracleUpdateFrequency` must be unique for each chainlink feed	Resolved
M-05	Escrow cannot bridge native assets	Resolved
M-06	`ZeroExAdapter` cannot process orders with `sellCurrency == localBuyCurrency`	Resolved
M-07	Missing slippage protection for Stargate swap in `OmnichainMessenger._distributeOrders()`	Resolved
M-08	Packed configs in `CommandLib` wrongly use 1 extra bit for custom flags	Resolved
M-09	Calling `approve(..., 0)` will revert for tokens that do not allow zero approvals	Acknowledged
M-10	Using `uint96` to store balances is incompatible for tokens with low value or high decimals	Resolved

ID	Description	Status
M-11	`msg.sender` is not validated for `ZeroExAdapter.phutureOnConsumeCallbackV1()`	Resolved
L-1	`BaseIndex.setConfig()` should accrue AUM fee when `metadata` address is updated	Resolved
L-2	`GovernanceProxy` does not restrict access to `receive()` function	Resolved
L-3	Hardcoded `sgReceive()` gas might not be sufficient for some chains	Resolved
L-4	`ChainRegistryLib` should limit `index` to 254	Resolved
L-5	Rounding in `OmnichainMessenger.pushIncomingOrders()` prevents removing currencies from anatomy	Resolved
L-6	`OrderBook.finishOrderExecution()` must not return empty `BoughtOrders`	Resolved
L-7	Calculation of `orderSellAmount` in `RebalancingLib.previewReserveRebalancingOrders()` should round down	Resolved
L-8	Price calculation in `ChainlinkLib` cannot handle tokens with large decimals	Acknowledged
L-9	Mapped transfers in `CommandLib._executeCommand()` could silently fail	Acknowledged
L-10	Don't reset `rebalancingFlags` when `BaseIndex.setConfig()` is called outside the context of `HomechainGovernance.finishRebalancing()`	Resolved
I-01	`Governance` is missing functions to access `setSequencerUptimeFeed()` and `withdrawCurrency()`	Resolved
I-02	Remove outer loop in `ConfigBuilder.getState()`	Resolved
I-03	Custom configuration for LayerZero	Acknowledged
I-04	LayerZero payload size limit	Acknowledged
I-05	Remove `index` from chains hash and currencies hash calculations	Resolved
I-06	`OmnichainMessenger` does not work on Aptos	Acknowledged
I-07	Lack of validation for Stargate chain paths	Acknowledged
I-08	Improve efficiency of trading algorithm in `RebalancingLib`	Resolved
I-09	`CurrencyLib` should implement code from latest solmate version	Resolved
I-10	`WORD_INDEX_MAX` constant in `BitSet` is redundant and can be removed	Resolved
I-11	`AnatomyValidationLib.validate()` should validate size of `currencyIndexSet`	Resolved
I-12	Missing validation of orders hash in `RemoteOmnichainMessenger.finishRebalancing()`	Resolved
I-13	Missing validation of `params.pendingOrderCounts.length` in `OrderBook.finishOrderExecution()`	Resolved

ID	Description	Status
I-14	`evm_version` is misspelt in `foundry.toml`	Resolved
I-15	Logic in `IndexCommandsLib.executeCommands()` can be simplified	Resolved
I-16	Use separate `additionalGas` values for each callback to allow for fine-grained gas control	Resolved
I-17	Performing `& type(uint32).max` before casting to `uint32` is redundant	Resolved
I-18	Optimization when iterating over chain IDs in a `BitSet`	Resolved
I-19	Preview functions in `ConfigBuilder.sol` shoud be `view` and `public`	Resolved
I-20	Missing `checkTotalWeight()` check in `ConfigBuilder.previewStartRebalancing()`	Resolved
I-21	Unncessary safe cast in `RebalancingLib.previewRebalancingOrders()`	Resolved
I-22	Pricing source difference between `HomechainOmnichainMessenger` and `Stargate`	Acknowledged
I-23	Optimizing gas at the expense of calldata size is more expensive on L2s	Acknowledged

4 Findings

High Risk

[H-01] Indexing issues in `RebalancingLib` lead to undefined rebalancing behavior

Context:

- RebalancingLib.sol#L100
- RebalancingLib.sol#L102
- RebalancingLib.sol#L105
- RebalancingLib.sol#L114
- RebalancingLib.sol#L205

Description: The `RebalancingLib` contains the `previewRebalancingOrders()` and `previewReserveRebalancingOrders()` functions. These functions generate the orders for the rebalancing and reserve rebalancing respectively.

It is critical that both the rebalancing and reserve rebalancing algorithms use the correct indexes. Failing to do so doesn't result in an immediate loss of funds, but the protocol ends up in an undefined state. For example, orders can be sent to the wrong chains and the rebalancing can get stuck. In the best case, only significant action by the `governance` could save the protocol.

Recommendation: Wrong indexes are used in multiple instances. The diff contains the fixes along with the reasoning for each instance.

```
## RebalancingLib.sol

@@ -29,6 +29,7 @@ library RebalancingLib {
         uint256 i;
         uint256 j;
         uint256 chainIndex;
+        uint256 activeChainIndex;
         uint256 currencyIndex;
         uint256 orderIndex;
         uint256 weightIndex;
@@ -97,12 +98,17 @@ library RebalancingLib {
             vars.activeChain = params.newAnatomy.chainIdSet.contains(vars.i);
             if (vars.activeChain || vars.priorChain) {
                 if (vars.priorChain) {
-                    chainOrders[vars.i].orders = new
OrderLib.Order[](params.orderCounts[vars.chainIndex]);
+                    // chainOrders stores the orders for the chains in the prior and
active anatomy
+                    // vars.orderIndex is how the union of prior and active chains
is indexed
+                    chainOrders[vars.orderIndex].orders = new
OrderLib.Order[](params.orderCounts[vars.chainIndex]);
                     uint256 count =
params.anatomy.currencyIdSets[vars.chainIndex].size();
-                    withdrawals[vars.i] =
IVault.CurrencyWithdrawal(BitSet.create(count), new uint96[](count));
+                    // need to use withdrawals[vars.orderIndex] with same reasoning
as above
```

```
+                    // BitSet needs to be created with size
params.currencies[vars.orderIndex].length since merely
+                    // the count of the currencies in the set doesn't specify the
range of IDs that the BitSet must hold
+                    withdrawals[vars.orderIndex] =
IVault.CurrencyWithdrawal(BitSet.create(params.currencies[vars.orderIndex].length),
new uint96[](count));
                }
-
-                IVault.CurrencyWithdrawal memory withdrawal = withdrawals[vars.i];
+                // need to use withdrawals[vars.orderIndex] with same reasoning as
above
+                IVault.CurrencyWithdrawal memory withdrawal =
withdrawals[vars.orderIndex];

                vars.currenciesHash = bytes32(0);
                Currency[] calldata currencies = params.currencies[vars.orderIndex];
@@ -111,7 +117,9 @@ library RebalancingLib {
                    vars.priorAsset = vars.priorChain &&
params.anatomy.currencyIdSets[vars.chainIndex].contains(vars.j);

                    vars.targetInBase = vars.activeChain
-                        &&
params.newAnatomy.currencyIdSets[vars.orderIndex].contains(vars.j)
+                        // using vars.orderIndex is wrong since the chains in the
newAnatomy are not a superset of the chains in the old anatomy
+                        // a new activeChainIndex must be introduced
+                        &&
params.newAnatomy.currencyIdSets[vars.activeChainIndex].contains(vars.j)
                        ?
valuationInfo.totalValuation.mulDivUp(params.newWeights[vars.weightIndex++],
MAX_WEIGHT)
                        : 0;

@@ -156,6 +164,7 @@ library RebalancingLib {

                unchecked {
                    if (vars.priorChain) ++vars.chainIndex;
+                    if (vars.activeChain) ++vars.activeChainIndex;
                    ++vars.orderIndex;
                }
            }
@@ -202,7 +211,9 @@ library RebalancingLib {
                    orders[0].orders[vars.weightIndex] = OrderLib.Order(
                        orderSellAmount,
                        OrderLib.OrderId(
-                            reserve, currencies[vars.j], currencies[vars.j],
params.chainIds[vars.orderIndex]
+                            // the wrong destination chain id is set
+                            // the correct chain id in this iteration is accessed
with vars.i
+                            reserve, currencies[vars.j], currencies[vars.j],
params.chainIds[vars.i]
                        )
                    );
```

Phuture: Fixed in PR 218

Renascence: The recommendation has been implemented.

[H-02] Rounding of sell amounts in `RebalancingLib` can make rebalancing stuck

Context:

- RebalancingLib.sol#L210
- RebalancingLib.sol#L261

Description: The `RebalancingLib` generates orders for the rebalancing and reserve rebalancing. It is important that orders with `localChain != destinationChain` and zero `sellAmount` do not increment `incomingOrders`.

This is because on the local chain, such orders will be skipped in `OrderLib.set()`.

```
## OrderLib.sol

function set(OrderRegistry storage self, Order[] calldata orders) internal {
    bytes32 newHash = self.ordersHash;
    for (uint256 i; i < orders.length; ++i) {
        if (orders[i].sellAmount == 0) continue;
        ...
```

As a result, the orders are never relayed to the destination chain and `incomingOrders` is not decremented. If `incomingOrders` doesn't reach zero, rebalancing cannot be finished.

An attacker can cause this scenario intentionally by setting the `reserve` balance to an amount that causes the sell amount to get rounded to zero. The most straightforward scenario is to front-run the rebalancing and leave just a dust amount of `reserve`.

Recommendation: There exist two instances of this issue in the `RebalancingLib`.

In `RebalancingLib.previewReserveRebalancingOrders()`, `reserveAmount > 0` does not imply `orderSellAmount > 0` and so the check needs to replace `reserveAmount` with `orderSellAmount`.

```
## RebalancingLib.sol

   unchecked {
-         if (vars.orderIndex != 0 && reserveAmount != 0) {
+         if (vars.orderIndex != 0 && orderSellAmount != 0) {
          ++orders[vars.orderIndex].incomingOrders;
       }
       vars.utilized += orderSellAmount;
```

In `RebalancingLib._createOrders()`, the check that `sellAmount > 0` is missing.

```
## RebalancingLib.sol

            // increment "fence" counter
-           if (buy.orderIndex != sell.orderIndex) {
+           if (buy.orderIndex != sell.orderIndex && sellAmount != 0) {
            ++orders[buy.orderIndex].incomingOrders;
          }
```

Phuture: Fixed in PR 218

Renascence: The recommendation has been implemented.

[H-03] Missing validation of `orderCounts` in `RebalancingLib.previewRebalancingOrders()` can make rebalancing stuck

Context:

- RebalancingLib.sol#L100

Description: The `StartRebalancingParams` struct that is passed to `RebalancingLib.previewRebalancingOrders()` contains an `orderCounts` array that specifies the number of orders for all the chains in the current (old) anatomy.

If `params.orderCounts[vars.chainIndex]` turns out to be too small, `RebalancingLib._createOrders()` will try to access an out-of-bounds index and revert. This is not an issue.

If on the other hand `params.orderCounts[vars.chainIndex]` is greater than the actual number of orders created, the `chainOrders` struct contains empty orders with all fields at the default zero value.

```
struct OrderId {
    Currency sellCurrency = 0;
    Currency localBuyCurrency = 0;
    Currency finalDestinationBuyCurrency = 0;
    uint256 finalDestinationChainId = 0;
}
```

The empty orders will be received on the remote chain, where the call to `RemoteOmnichainMessenger._injectLocalBuyCurrency()` inside `RemoteOmnichainMessenger.pushOrders()` will revert since there is no bridging info for `finalDestinationChainId = 0`.

This is a vulnerability that needs to be fixed since by front-running the rebalancing, a user can influence the amount of orders created.

Recommendation: The solution incurs some overhead as the number of orders is only known once the last chain has been iterated over. All prior chains can have new orders until the last call to `RebalancingLib._createOrders()` has been executed.

It is possible to iterate over all orders from `chainOrders[0]` to `chainOrders[vars.orderIndex-1]` and ensure that `chainOrders[i].orders.length == vars.counters[i]`.

```
## RebalancingLib.sol

         }

+        for (uint i; i < vars.orderIndex; i++) {
+            require(chainOrders[i].orders.length == vars.counters[i]);
+        }
+
         if (params.newWeights.length != vars.weightIndex) revert UnutilizedWeights();
         if (vars.chainsHash != chainsHash) revert ChainsHashMismatch();
     }
```

Phuture: Fixed in PR 233

Renascence: The check has been implemented in `ConfigBuilder.sol`. The expected order count for each chain must match the actual order count, as indicated by its `counters` value.

[H-04] Indexing issues in `ConfigBuilder` lead to corrupted anatomy data when rebalancing is finished

Context:

- ConfigBuilder.sol#L186-L187
- ConfigBuilder.sol#L275
- ConfigBuilder.sol#L301
- ConfigBuilder.sol#L311-L312

Description: The wrong indexing is used in `ConfigBuilder.startRebalancing()` and `ConfigBuilder.finishRebalancing()`.

In the `ConfigBuilder.startRebalancing()` function the wrong indexing simply causes the function to revert in some cases since `chainOrders[chainIndex]` accesses `chainOrders` with an out-of-bounds index.

On the other hand, in the `ConfigBuilder.finishRebalancing()` function, wrong `results` can be assigned to the data structures that keep track of the new anatomy. This not only makes rebalancing get stuck, it also corrupts the protocol's data.

Recommendation: The diff contains the fixes along with the reasoning for each instance.

```
## ConfigBuilder.sol

@@ -29,6 +29,9 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {
         Anatomy anatomy;
         uint256[] chainIds;
         a160u96[][] currencies;
+        uint256[] resultIndexes;
+        uint256 resultIndexBit;
     }

     // `reserve` is registered as first currency during `Index` deployment
@@ -183,8 +186,9 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {
             for (uint256 b = BitSet.find(w, 0); BitSet.hasNext(w, b);) {
                 uint256 chainIndex = BitSet.valueAt(0, b);
                 if (chainIndex != HOME_CHAIN_INDEX) {
-                    chainOrdersHash[i++] =
keccak256(abi.encode(chainOrders[chainIndex]));
-                    emit RemoteChainOrders(params.chainIds[chainIndex],
chainOrders[chainIndex]);
+                    // chainIndex is the ID of the chain
+                    // say we have IDs 0 and 2, then we need to access
chainOrders[0] and chainOrders[1],
+                    // not chainOrders[0] and chainOrders[2]
+                    chainOrdersHash[i] = keccak256(abi.encode(chainOrders[i+1]));
+                    emit RemoteChainOrders(params.chainIds[chainIndex],
chainOrders[i+1]);
+                    i++;
                 }
```

```
                unchecked {
                    b = BitSet.find(w, b + 1);
@@ -264,6 +268,7 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {

         FinishRebalancingVars memory vars;
         vars.anatomy.chainIdSet =
BitSet.create(MAX_CHAIN_INDEX_COUNT).add(HOME_CHAIN_INDEX);
+        // a new set is needed to keep track of the indexes for the results array
+        // that point to results for the new anatomy
+        vars.resultIndexes =
BitSet.create(MAX_CHAIN_INDEX_COUNT).add(HOME_CHAIN_INDEX);

         uint256 i;
         // chain id set contains single word
@@ -272,7 +277,10 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {
             uint256 chainId = BitSet.valueAt(0, b);
             if (keccak256(abi.encode(results[i])) != resultHashOf[chainId]) revert
ResultHash();

-            if (results[i].currencies.length != 0)
vars.anatomy.chainIdSet.add(chainId);
+            if (results[i].currencies.length != 0) {
+                vars.anatomy.chainIdSet.add(chainId);
+                vars.resultIndexes.add(i);
+            }

             emit FinishChainRebalancing(
                 results[i].chainId, results[i].snapshot, results[i].currencyIdSet,
results[i].currencies
@@ -298,7 +306,8 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {
         w = vars.anatomy.chainIdSet[0];
         for (uint256 b = BitSet.find(w, 0); BitSet.hasNext(w, b);) {
             uint256 chainIndex = BitSet.valueAt(0, b);
-            vars.chainIds[i] = results[chainIndex].chainId;
+            // the next chainId is the chainId of the next result in the new anatomy
+            vars.chainIds[i] = results[vars.resultIndexBit].chainId;

             if (chainIndex == HOME_CHAIN_INDEX) {
                 (vars.currencies[HOME_CHAIN_INDEX], vars.anatomy.currencyIdSets[i])
= _insertReserveIfNeeded(
@@ -308,13 +317,14 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {
                 homeCurrencies = vars.currencies[HOME_CHAIN_INDEX];
                 latestSnapshot = results[HOME_CHAIN_INDEX].snapshot;
             } else {
-                vars.anatomy.currencyIdSets[i] = results[chainIndex].currencyIdSet;
-                vars.currencies[i] = results[chainIndex].currencies;
+                // populate new anatomy with the results in the new anatomy
+                vars.anatomy.currencyIdSets[i] =
results[vars.resultIndexBit].currencyIdSet;
+                vars.currencies[i] = results[vars.resultIndexBit].currencies;
             }

             unchecked {
                 ++i;
                 b = BitSet.find(w, b + 1);
+                vars.resultIndexBit = BitSet.find(vars.resultIndexes[0],
vars.resultIndexBit + 1);
             }
         }
```

Phuture: Fixed in PR 231 and PR 240

Renascence: The recommendation has been implemented.

[H-05] `RedstoneChainlinkPriceOracle.valuation()` can return outdated valuation and funds can be stolen from the index

Context:

- RedstoneChainlinkPriceOracle.sol#L45

Description: The `CalldataExtractor.extractTimestampsAndAssertAllAreEqual()` function only checks the calldata format but not the legitimacy of the data or the timestamp.

This means that if the oldest saved price in `staticRedstoneData` is say from 1 hour ago or even a week ago, an attacker can provide `bytes calldata` such that the outdated valuation is returned.

An attacker can intentionally access the outdated valuation to `deposit()` into the Index and receive more shares than he is entitled to or `redeem()` from the Index and receive more constituents/reserve than he should be able to.

This is a direct loss to the other users of the protocol.

Recommendation: The `else` case in the `RedstoneChainlinkPriceOracle.valuation()` function needs to validate the `staticRedstoneData.timestamp`. If the timestamp is outdated, the function needs to revert and the outdated valuation can thus not be used.

```
## RedstoneChainlinkPriceOracle.sol

          staticRedstoneData.valuation = redstoneValuation.safeCastTo128();
          lastRedstoneData = staticRedstoneData;
      } else {
+         RedstoneDefaultsLib.validateTimestamp(staticRedstoneData.timestamp);
          _reservePrice = ChainlinkLib.price(info.decimals[0],
info.chainlinkAggregators[0], maxOracleUpdateFrequency);
          _valuation = info.balances[0].convertToBaseDown(_reservePrice);
```

Phuture: Fixed in PR 218

Renascence: The recommendation has been implemented.

[H-06] `retryRedeem()` can be used to block the LayerZero pathway

Context:

- HomechainOmnichainMessenger.sol#L104

Description: There is no validation of bytes passed as `options` to `_lzSend()` inside the `retryRedeem()` function. An attacker can encode the adapterParams and specify a very small gasLimit for the relayer to deliver the message to the destination chain.

A transaction with such a small gas limit would revert inside the lzReceive() function due to out-of-gas exception.

As a consequence, it would end up as `storedPayload` inside the Endpoint contract.

This would block the delivery of all the subsequent messages until the payload is cleared.

Recommendation: There should be validation that a certain minimum gas is specified while leaving the user an option of airdropping tokens to the destination chain. See adapterParams for more details.

```
## HomechainOmnichainMessenger.sol

@@ -95,6 +95,10 @@ contract HomechainOmnichainMessenger is
IHomechainOmnichainMessenger, OmnichainM
             bytes memory path = _getPathOrRevert(retry.eid);

             for (uint256 j; j < retry.callbacks.length; ++j) {
+
+                uint256 minGasAmountForChain; // read from configuration
+                validateOptions(retry.options[j], minGasAmountForChain);
+
                 _lzSend(
                     retry.eid,
                     path,
@@ -110,6 +114,18 @@ contract HomechainOmnichainMessenger is
IHomechainOmnichainMessenger, OmnichainM
         if (balance != 0) _refund(refundAddress, balance);
     }

+    function validateOptions(bytes memory options, uint256 minGasAmountForChain)
internal {
+        if (!(options.length == 34 || options.length > 66)) revert Invalid();
+        uint16 txType;
+        uint extraGas;
+        assembly {
+            txType := mload(add(options, 2))
+            extraGas := mload(add(options, 34))
+        }
+        if (minGasAmountForChain > extraGas) revert Invalid();
+        if (!(txType == 1 || txType == 2)) revert Invalid();
+    }
+
```

Phuture: Fixed in PR 236

Renascence: The recommendation has been implemented.

[H-07] ETH balances of home and remote messenger can be drained

Context:

- HomechainOmnichainMessenger.sol#L80
- RemoteOmnichainMessenger.sol#L65
- RemoteOmnichainMessenger.sol#L101

Description: During the `executeOrder()` function execution, in case the `orderId.finalDestinationChainId != block.chainid`, the order is outgoing and the recipient of the token is the messenger contract. If the output token after the trade is `ETH` it will end up as the balance of the `HomechainOmnichainMessenger` or `RemoteOmnichainMessenger` contract.

```
## OrderBook.sol

bool isOutgoing = orderId.finalDestinationChainId != block.chainid;

address recipient = isOutgoing ? messenger : vault;
```

Looking at the `_distributeOrders()` function, there is the intention to transfer native with Stargate:

```
## OmnichainMessenger.sol

if (pendingOrder.currency.isNative()) sgMessageFee += pendingOrder.totalBought;
```

Any native balance in the messenger can be drained through several functions.

An example is the `retryRedeem()` function that calls the underlying `_lzSend()` internal function which uses `address(this).balance` to pay for the `lzEndpoint.send()` function call. If `retryRedeem()` is called with `msg.value = 0` it will be using the balance of the messenger contract to pay LayerZero fees and the remaining balance will be transferred to the user.

This issue is present in multiple functions.

User-facing functions can be used straight up to drain the messenger balance:

- `HomechainOmnichainMessenger.retryRedeem()`
- `HomechainOmnichainMessenger.redeem()` (cannot be called during rebalancing so it's not an actual issue)

Governance-restricted functions are problematic if `governance` does not send along sufficient native funds such that the LayerZero fees cut into the funds that are meant to be swapped to another chain.

- `HomechainOmnichainMessenger.broadcastOrders()`
- `HomechainOmnichainMessenger.broadcastReserveOrders()`
- `RemoteOmnichainMessenger.sendCurrenciesHash()`
- `RemoteOmnichainMessenger.finishRebalancing()`

Recommendation: The recommended fix for the user-facing functions:

```
## HomechainOmnichainMessenger.sol

@@ -64,6 +64,7 @@ contract HomechainOmnichainMessenger is
IHomechainOmnichainMessenger, OmnichainM
         IIndex.RedemptionCommandParams calldata commandParams,
         SendParams calldata sendParams
     ) external payable {
+        uint256 balanceBefore = address(this).balance - msg.value;
         if (keccak256(abi.encode(sendParams.config)) != layerZeroConfigHash) revert
ConfigHash();
         if (sendParams.batches.length != sendParams.config.eIds.length()) revert
InvalidBatches();

@@ -76,8 +77,8 @@ contract HomechainOmnichainMessenger is
IHomechainOmnichainMessenger, OmnichainM
             }
         }

-        uint256 balance = address(this).balance;
-        if (balance != 0) _refund(payable(msg.sender), balance);
+        uint256 refund = address(this).balance - balanceBefore;
+        if (refund != 0) _refund(refundAddress, refund);
     }

     function retryRedeem(
@@ -86,6 +87,7 @@ contract HomechainOmnichainMessenger is
IHomechainOmnichainMessenger, OmnichainM
         address payable refundAddress,
         address zroPaymentAddress
     ) external payable {
+        uint256 balanceBefore = address(this).balance - msg.value;
         for (uint256 i; i < params.length; ++i) {
             IIndex(index).redeemK(params[i], msg.sender);

-        uint256 balance = address(this).balance;
-        if (balance != 0) _refund(refundAddress, balance);
+        uint256 refund = address(this).balance - balanceBefore;
+        if (refund != 0) _refund(refundAddress, refund);
+    }
```

Fixing governance-restricted functions is optional as governance is fully trusted.

Phuture: Fixed in PR 240

Renascence: The recommendation has been implemented.

[H-08] Small sell orders cannot be executed and settled across chains

Context:

- OrderBook.sol#L106-#L147

Description: As there is no minimum sell amount threshold, orders with 1 wei of sell amount can be created. During order execution, if the price of the asset that is being sold is smaller than the asset being bought, the only way of executing this order is buying 0 amount. This leads to issues downstream. If this is the only sell order to generate a buy order for the remote chain, the `finishOrderExecution()` function will generate a `PendingOrder` with `totalBought` equal to zero.

```
## OrderBook.sol

    bytes32 idKey = orderId.id();
    uint256 bought = _balancesOf[idKey];
>   pendingOrders[chainIndex].totalBought += bought;
    pendingOrders[chainIndex].orders[orderIndex] = BoughtOrder(bought,
orderId.finalDestinationBuyCurrency);
```

This will revert inside the `OmnichainMessenger._distributeOrders()` function as Stargate swap can't send zero amounts.

In turn, the rebalancing cannot be finalized on the remote chain as it expects incoming orders.

Recommendation: Not handling small amounts is a broad architectural problem that spans several other issues. One recommendation for handling it is defaulting to sending a LayerZero message in case the Stargate swap cannot be executed so the `incomingOrders` of the destination chain get decremented and rebalancing can be finalized.

Phuture: Fixed in PR 225

Renascence: The issue has been fixed by allowing to decrement incoming orders via LayerZero message.

[H-09] `depositWithCommand()` undercounts `deposited` when `reserve` is ETH

Context:

- IndexCommandsLib.sol#L21-L23

Description: `IndexCommandsLib.depositWithCommand()` has the following logic:

1. Records the index's current reserve and currency balance:

```
uint256 reserveBefore = reserve.balanceOfSelf();
uint256 balanceBefore = commandParams.currency.balanceOfSelf();
if (commandParams.currency.isNative()) balanceBefore -= msg.value;
```

2. Transfers currency from the user into the contract and executes user-specified commands:

```
commandParams.currency.selfDeposit(params.amount);

CommandLib.callCommand(
    commandParams.command,
    params.config.shared.metadata,
    CommandLib.BalanceState(commandParams.currency, balanceBefore)
);
```

3. Calculates `deposited` as the difference between the reserve balance after and the recorded reserve balance:

```
if (commandParams.currency.balanceOfSelf() < balanceBefore) revert
InvalidDepositData();

deposited = uint96(reserve.balanceOfSelf() - reserveBefore);
```

Since it is possible for reserve or currency to be native tokens, `msg.value` is subtracted from `balanceBefore` when `commandParams.currency` is native tokens. This ensures that the difference between `commandParams.currency.balanceOfSelf()` and `balanceBefore` afterwards will include `msg.value`.

However, `msg.value` is not subtracted from `reserveBefore`, which means that any ETH sent along with the call will be included in `reserveBefore`. Consequently, the ETH sent will then be excluded from `deposited`, which is problematic as `deposited` represents the amount of funds deposited by the user into the index.

This will undercount the user's deposit when `reserve` is native tokens, for example:

- Assume that `reserve` is ETH, and a user wants to deposit 1 ETH and 2000 USDC in one call.
- He calls `Index.depositWithCommand()` with:
  - `msg.value = 1 ether`
  - `commandParams.currency` as USDC, and `params.amount = 2000e6`
- In `IndexCommandsLib.depositWithCommand()`:
  - `reserveBefore = 1 ether`, since it includes `msg.value`.
  - 2000 USDC is transferred into the contract and swapped to 1 ETH.
  - Afterwards, `reserve.balanceOfSelf() = 2 ether`, so `deposited = 1 ether`.
- As a result, his deposit is recorded as only 1 ETH, causing him to lose the other 1 ETH.

Recommendation: Subtract `msg.value` from `reserveBefore` when `reserve` is ETH:

```
  uint256 reserveBefore = reserve.balanceOfSelf();
  uint256 balanceBefore = commandParams.currency.balanceOfSelf();
+ if (reserve.isNative()) reserveBefore -= msg.value;
  if (commandParams.currency.isNative()) balanceBefore -= msg.value;
```

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented. Currency balances are now fetched with a new `_getBalance()` function, which subtracts `msg.value` from `balanceOfSelf()` if the currency is native ETH.

[H-10] Reentrancy via callback from `SKIM` command leads to theft of `Vault` funds

Context:

- Index.sol#L68
- L2Index.sol#L73
- Vault.sol#L134-L142
- Validator.sol#L44-L53

Description: An attacker can get a callback from the `CommandLib._call()` function, since via the `SKIM` command, an untrusted contract can be called if `currencyState.curreny` is the `NATIVE` currency.

```
## Validator.sol

    if (mapType == SKIM) {
        if (target.addr() != Currency.unwrap(currencyState.currency)) revert
UnsupportedTarget();
        (, address to) = abi.decode(data, (uint8, address));
        uint256 amount = currencyState.currency.balanceOf(msg.sender) -
currencyState.minBalance;
        if (currencyState.currency.isNative()) {
>           targetInfo.target = to;
            targetInfo.value = amount;
```

`IndexCommandsLib` iterates over all currencies in the `snapshot` and so in the downstream `SKIM` command there will be more currencies processed after the untrusted contract has received the callback.

The dangerous line is the following: `uint256 amount = currencyState.currency.balanceOf(msg.sender) - currencyState.minBalance;`.

By depositing or donating to the `Vault`, an attacker can increase the `Vault`'s balance, such that the `SKIM` command sends out more funds than it should. This is a direct theft of funds and there exist different paths to achieve the same outcome.

- `Index.redeemK()` / `L2Index.redeemK()` → `SKIM` Native → `Index.deposit()` / `Index.depositWithCommand()`
- `Index.redeemK()` / `L2Index.redeemK()` → `SKIM` Native → `OmnichainMessenger.pushIncomingOrders()` → `Vault.donate()`

Recommendation: To address all reentrancies that a regular user can exploit, three additional reentrancy guards must be applied.

```
## Index.sol

-    function redeemK(RedeemKParams calldata params, address forwardedSender)
external {
+    function redeemK(RedeemKParams calldata params, address forwardedSender)
external nonReentrant {


## L2Index.sol

-    function redeemK(RedeemKParams calldata params, address forwardedSender)
external {
+    function redeemK(RedeemKParams calldata params, address forwardedSender)
external nonReentrant {


## Vault.sol

-    function donate(Currency currency, bytes memory data) external only(orderBook) {
+    function donate(Currency currency, bytes memory data) external nonReentrant
only(orderBook) {
```

Note that since the `governance` and `fundManager` role are considered trusted, reentrancies for these roles are not considered an issue and are not addressed.

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented. Note that the `nonReentrant` modifier for `redeemK()` was added to the internal `_redeemK()` function instead.

[H-11] Stargate can not transfer asset amounts smaller than the conversion rate for the pool

Context:

- OmnichainMessenger.sol#L172

Description: Stargate pools have a concept of convert rate. It's calculated based on the `sharedDecimals` and `localDecimals` for a specific pool. For example, the `DAI` Pool has the `sharedDecimals` set to 6 while `localDecimals` is 18.

The convert rate is then: `10^(localDecimals - sharedDecimals) = 10^12`.

Here is the DAI Pool on Ethereum and the convert rate logic inside the Pool contract.

Transferring an amount less than `10^12` on a pool with a convert rate different than 1 with Stargate is not possible: https://github.com/stargate-protocol/stargate/blob/5f0dfd2/contracts/Router.sol#L126. The amount needs to be a multiple of `10^12` and any dust amount is not transferred.

Looking at reserve rebalancing, `incomingOrders` for a chain are created even if the amounts are small: https://github.com/Phuture-Finance/phuture-v2-contracts/blob/0ee3b0b909d0c9795ffefdd377b8671888c3a85d/src/libraries/RebalancingLib.sol#L211.

During this first step of reserve rebalancing, incoming orders can be created and sent to a remote chain while the size of the trade is less than `10^12`.

After all the orders are executed on the home chain, `finishRebalancing()` will be called to send the actual orders to all the remote chains. An order smaller than `10^12` can never be sent through Stargate.

This is an issue during reserve rebalancing and regular rebalancing.

Although it is even more likely to occur during the normal rebalancing on the remote chains, the following scenario highlights that there is a griefing concern as well.

In normal circumstances, reserve rebalancing will only occur if there is sufficient reserve so the issue for small amounts may not seem realistic.

However, an adversary can front-run the transaction that calls `startReserveRebalancing()` and redeem his shares from the reserve to intentionally leave a small amount of reserve to rebalance.

Recommendation: Not handling small amounts is a broad architectural problem that spans several other issues and was already mentioned in **H-08**. The solution to **H-08** will fix this issue as well.

Phuture: Fixed in PR 225

Renascence: The issue has been fixed by checking if the `totalBought` amount is equal to or higher than the `convertRate` for the Stargate Pool. We advise for extra safety to ask the Stargate team if this condition is sufficient to always make a swap.

Medium Risk

[M-01] `send()` doesnt work on zkSync

Context:

- HomechainOmnichainMessenger.sol#L282

Description: `send()` and `transfer()` don't work on zkSync due to forwarding fixed gas (2300). Moreover, their use is discouraged on other chains as gas costs can change.

See these resources:

- https://twitter.com/zksync/status/1644139364270878720
- https://consensys.io/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Recommendation: Use the low-level call:

```
## HomechainOmnichainMessenger.sol

     function _refund(address payable recipient, uint256 amount) internal {
-        if (!recipient.send(amount)) revert RefundFailed();
+        (bool success, ) = recipient.call{value: amount}("");
+        if (!success) revert RefundFailed();
     }
```

Phuture: Fixed in PR 219

Renascence: The recommendation has been implemented.

[M-02] Tokens with non-string metadata revert

Context:

- OmnichainMessenger.sol#L204-L205

Description: Some tokens(e.g. MKR) have metadata fields (name/symbol) encoded as bytes32 instead of the string prescribed by the ERC20 specification.

Calls to name and symbol will revert.

Recommendation: The following fix provides compatibility for both string and bytes32:

```
## OmnichainMessenger.sol

@@ -29,6 +29,8 @@ abstract contract OmnichainMessenger is BlockingApp,
IStargateReceiver, IOmnicha
         uint8 decimals;
     }

+    error QueryFailed();
+
     /// @dev update currencies hash on home chain
     uint8 internal constant REGISTER_CURRENCY = 0;
     /// @dev if reserve changed we only send redeemedK, recipient and owner to each
remote chain
```

```
@@ -201,8 +203,8 @@ abstract contract OmnichainMessenger is BlockingApp,
IStargateReceiver, IOmnicha
                    ICurrencyMetadata.CurrencyMetadata(_info.name, _info.symbol,
_info.decimals, currency, info.ids[i]);
             } else {
                 result.metadata[i] = ICurrencyMetadata.CurrencyMetadata(
-                    IERC20Metadata(Currency.unwrap(currency)).name(),
-                    IERC20Metadata(Currency.unwrap(currency)).symbol(),
+                    name(currency),
+                    symbol(currency),
                     IERC20Metadata(Currency.unwrap(currency)).decimals(),
                     currency,
                     info.ids[i]
@@ -229,4 +231,32 @@ abstract contract OmnichainMessenger is BlockingApp,
IStargateReceiver, IOmnicha
             }
         }
     }
+
+    function name(Currency currency) internal view returns (string memory) {
+        return queryStringOrBytes32(Currency.unwrap(currency),
IERC20Metadata.name.selector);
+    }
+
+    function symbol(Currency currency) internal view returns (string memory) {
+        return queryStringOrBytes32(Currency.unwrap(currency),
IERC20Metadata.symbol.selector);
+    }
+
+    function queryStringOrBytes32(address target, bytes4 selector) internal view
returns (string memory s) {
+        (bool success, bytes memory returndata) =
target.staticcall(abi.encodeWithSelector(selector));
+        if (!success) revert QueryFailed();
+
+        if (returndata.length == 32) { // returned data is bytes32
+            s = string(returndata);
+
+            // Find last non-zero byte
+            uint256 length = 32;
+            for (; length != 0; length--) if (returndata[length - 1] != 0) break;
+
+            // Shorten length of string
+            assembly {
+                mstore(s, length)
+            }
+        } else { // returned data is string
+            s = abi.decode(returndata, (string));
+        }
+    }
 }
```

Phuture: Fixed in PR 230

Renascence: The recommendation has been implemented.

[M-03] `ConfigBuilder._prepareWithdrawals()` **assumes that** `reserve` **is always sold and never bought**

Context:

- ConfigBuilder.sol#L465

Description: The logic in `ConfigBuilder._prepareWithdrawals()` assumes that `withdrawals[0].amounts[0]` is the `reserve` amount. The `reserve` is only in this location `if (currentInBase > targetInBase)` in the `RebalancingLib`.

```
## RebalancingLib.sol

} else if (vars.currentInBase > vars.targetInBase) {
    uint96 assets = valuationInfo.balances[vars.currencyIndex].mulDivDown(
        vars.currentInBase - vars.targetInBase, vars.currentInBase
    ).safeCastTo96();
    vars.sellDeltas = vars.sellDeltas.push(
        vars.currentInBase - vars.targetInBase,
        assets,
        vars.orderIndex,
        currencies[vars.j],
        valuationInfo.prices[vars.currencyIndex]
    );


    withdrawal.amounts[withdrawal.currencyIndexSet.size()] = assets;
    withdrawal.currencyIndexSet.add(vars.j);
}
```

In a rebalancing scenario where the weight of the `reserve` is increased, this assumption does not hold and it's possible that `reserve` is bought (`currentInBase !> targetInBase`).

It is unlikely that `reserve` needs to be bought but the function still needs to consider the possibility.

Recommendation: If the `homeWithdrawal.currencyIndexSet` does not contain the `RESERVE_INDEX`, the correct behavior is to skip the `reserve` specific logic in `ConfigBuilder._prepareWithdrawals()`.

```
## ConfigBuilder.sol

     function _prepareWithdrawals(IVault.CurrencyWithdrawal[] memory withdrawals,
uint96 reserveBalance) internal pure {
         // reserve is always presented in withdrawals as it should be totally sold
         IVault.CurrencyWithdrawal memory homeWithdrawal = withdrawals[0];
-        homeWithdrawal.amounts[RESERVE_INDEX] -= uint96(Math.min(reserveBalance,
homeWithdrawal.amounts[RESERVE_INDEX]));
-        if (homeWithdrawal.currencyIndexSet.size() == 1 &&
homeWithdrawal.amounts[RESERVE_INDEX] == 0) {
-            homeWithdrawal.currencyIndexSet.remove(RESERVE_INDEX);
+        if (homeWithdrawal.currencyIndexSet.contains(RESERVE_INDEX)) {
+            homeWithdrawal.amounts[RESERVE_INDEX] -= uint96(Math.min(reserveBalance,
homeWithdrawal.amounts[RESERVE_INDEX]));
+            if (homeWithdrawal.currencyIndexSet.size() == 1 &&
homeWithdrawal.amounts[RESERVE_INDEX] == 0) {
+                homeWithdrawal.currencyIndexSet.remove(RESERVE_INDEX);
+            }
         }

         IVault.CurrencyWithdrawal memory emptyWithdrawal;
```

Phuture: Fixed in PR 223

Renascence: The recommendation has been implemented.

[M-04] `maxOracleUpdateFrequency` must be unique for each chainlink feed

Context:

- ChainlinkPriceOracle.sol#L17
- RedstoneChainlinkPriceOracle.sol#L36

Description: The `maxOracleUpdateFrequency` is currently an immutable variable for each `ChainlinkPriceOracle` / `RedstoneChainlinkPriceOracle`. It is used to determine when a Chainlink feed has timed out.

However, different Chainlink feeds have different timeouts and the `maxOracleUpdateFrequency` must be set to the maximum of all timeouts.

An oracle with a lower timeout is not recognized as stale until the full `maxOracleUpdateFrequency` duration has passed.

In such a case, an attacker could make use of the outdated price to mint Index shares at a discount or to receive too many funds from redemption.

Recommendation: Parameters that are unique to each Chainlink feed are stored in the price source mapper contracts (`ChainlinkPriceSourceMapper`, `RedstoneChainlinkPriceSourceMapper`). This is a natural place to add the `updateFrequency` for each Chainlink feed.

The `CurrencyInfo` and `PriceOracleInfo` structs should be extended such that they can store the `updateFrequency` for each Chainlink feed.

```
## ChainlinkPriceSourceMapper.sol

    struct CurrencyInfo {
        ChainlinkLib.Aggregator[] aggregators;
        uint96 balance;
        uint8 decimals;
+       uint256 updateFrequency;
    }


## RedstoneChainlinkPriceSourceMapper.sol

    struct PriceOracleInfo {
        uint256[] sourceSet;
        uint96[] balances;
        uint8[] decimals;
        ChainlinkLib.Aggregator[][] chainlinkAggregators;
        bytes32[] redstoneFeedIds;
+       uint256[] updateFrequencies;
    }
```

Chainlink feeds and their update frequencies can be found here.

Phuture: Fixed in PR 222 and PR 235.

Renascence: The issue has been fixed by adding the `staleAfter` field to the `Aggregator` struct, such that each Chainlink feed has its unique `staleAfter` value.

[M-05] Escrow cannot bridge native assets

Context:

- Escrow.sol#L106

Description: The `Escrow` contract has the functionality of transferring assets via Stargate after all the trading has finished. The issue with the Stargate bridging logic is that it is not able to send ETH. This is problematic on Linea as it only supports the ETH pool: `https://stargateprotocol.gitbook.io/stargate/developers/pool-ids`

Recommendation: To enable sending ETH, add the ETH accumulated after trading to `msg.value` if the currency being bridged is native.

```
## Escrow.sol

@@ -92,8 +92,6 @@ contract Escrow is IPhutureOnMessageCallback {
     function _bridge(BridgeParams memory params) internal {
         uint256 amount = params.currency.balanceOfSelf();

-        params.currency.approve(params.sgRouter, amount);
-
         IStargateRouter.lzTxObj memory lzTxObj;
         (uint256 sgMessageFee,) = IStargateRouter(params.sgRouter).quoteLayerZeroFee(
             params.dstEid,
@@ -103,6 +101,10 @@ contract Escrow is IPhutureOnMessageCallback {
             lzTxObj
         );

+        if (params.currency.isNative()) sgMessageFee += amount;
+
+        params.currency.approve(params.sgRouter, amount);
+
         try IStargateRouter(params.sgRouter).swap{value: sgMessageFee}(
             params.dstEid,
             params.srcPoolId,
```

Phuture: Fixed in PR 241

Renascence: The recommendation does not properly address this issue since `amount` is the balance of the `Escrow` contract, and so `amount + sgMessageFee` is more than the contract's balance and cannot be sent to the `StargateRouter`. The fix that the team has implemented correctly bridges `amount - sgMessageFee` while sending `amount` to the `StargateRouter`.

[M-06] `ZeroExAdapter` cannot process orders with `sellCurrency == localBuyCurrency`

Context:

- ZeroExAdapter.sol#L35
- OrderBook.sol#L121-L127

Description: Outgoing orders can have `sellCurrency == localBuyCurrency` and `ZeroExAdapter`, which is the only contract that implements the `phutureOnConsumeCallbackV1()` function, is not able to deal with this scenario.

```
## ZeroExAdapter.sol

params.currencyOut.transfer(params.recipient, params.currencyOut.balanceOfSelf() -
balanceBefore);
```

No funds are transferred to the `recipient` and the slippage check in `OrderBook` fails.

Recommendation: The client has discovered and fixed the issue during the audit by implementing the `phutureOnConsumeCallbackV1()` function in `OmnichainMessenger`. No further recommendation is needed.

Phuture: Fixed in PR 219

Renascence: The issue has been fixed.

[M-07] Missing slippage protection for Stargate swap in `OmnichainMessenger._distributeOrders()`

Context:

- OmnichainMessenger.sol#L178

Description: The Stargate swap in `OmnichainMessenger._distributeOrders()` is executed without slippage protection.

```
## OmnichainMessenger.sol

    IStargateRouter(stargateComposer).swap{value: sgMessageFee}(
        dstEid,
        sgPoolIds.src,
        sgPoolIds.dst,
        payable(address(this)),
        pendingOrder.totalBought,
>       0,
        lzTxObj,
        to,
        payload
    );
```

Recommendation: Since `OmnichainMessenger._distributeOrders()` is only called in a trusted context by the `governance`, it is possible to pass the `minAmountLD` values as a parameter to the function and to calculate it off-chain.

Phuture: Fixed in PR 230

Renascence: The recommendation has been implemented and the `minAmountLD` parameter is now part of the `SgParams` struct.

[M-08] Packed configs in `CommandLib` wrongly use 1 extra bit for custom flags

Context:

- CommandLib.sol#L31-L32
- CommandLib.sol#L65

Description: In `CommandLib`, commands are specified using packed configs of 20 bits:

```
uint256 internal constant PACKED_CONFIG_TARGET_CONFIG_BITS = 5;
uint256 internal constant PACKED_CONFIG_SIZE_BITS = 20;
```

In each packed config, the first 5 bits are used for custom flags, leaving only 15 bits for the currency index.

However, in `_executeCommand()`, the currency index is read from each packed config as such:

```
int256 currencyIndex = uint16(pc » PACKED_CONFIG_TARGET_CONFIG_BITS) - 1;
```

`pc` is shifted right by 5 bits, and then the next 16 bits are read. This causes 1 extra bit to be read from the next packed config. If the first custom flag (LSB) for the next packed config is set, `currencyIndex` will include this 1 extra bit, causing it to be wrong.

For example:

```
function testPackedConfig() public {
    uint256 index = 1;
    uint256 flags = 1; // 00001
    uint256 config = (index « PACKED_CONFIG_TARGET_CONFIG_BITS) | flags; // ...0100001

    uint256 packedConfig = config « PACKED_CONFIG_SIZE_BITS | config;

    uint256 currencyIndex = uint16(packedConfig » PACKED_CONFIG_TARGET_CONFIG_BITS) -
1;
    console2.log(currencyIndex); // 32768 instead of 0
}
```

The impact is that `redeem()` and `redeemK()` in the `Index` contract will unexpectedly revert when custom flags are used since the code will perform an out-of-bounds access to the `currencyStates` array.

Recommendation: Use 4 bits for custom flags instead, leaving 16 bits for the currency index:

```
- uint256 internal constant PACKED_CONFIG_TARGET_CONFIG_BITS = 5;
+ uint256 internal constant PACKED_CONFIG_TARGET_CONFIG_BITS = 4;
```

Phuture: Fixed in PR 223

Renascence: The recommendation has been implemented.

[M-09] Calling `approve(..., 0)` will revert for tokens that do not allow zero approvals

Context:

- CommandLib.sol#L105
- ZeroExAdapter.sol#L33
- Escrow.sol#L88
- Escrow.sol#L118
- OmnichainMessenger.sol#L184

Description: Throughout the code, approvals to external contracts are implemented using the following pattern:

```
bool approve = allowanceTarget != address(0);
if (approve) currency.approve(allowanceTarget, type(uint256).max);

(success, returnData) = target.call{value: value}(data);
if (!success) revert CallFailed(returnData);

if (approve) currency.approve(allowanceTarget, 0);
```

Using `CommandLib._call()` as an example, the pattern is:

1. Grant approval to the external contract.
2. Perform the external call.
3. Call `approve(..., 0)` to revoke any remaining allowance.

All instances of this pattern have been listed above.

However, calling `approve(..., 0)` will always revert for tokens that do not allow zero approvals.

The most prominent example is BNB:

```
function approve(address _spender, uint256 _value)
    returns (bool success) {
    if (_value <= 0) throw;
    allowance[msg.sender][_spender] = _value;
    return true;
}
```

This makes the protocol incompatible with BNB in various parts of the code.

For example, when `IndexCommandsLib.executeCommands()` is used to perform a call to whitelisted exchange proxies with BNB in `depositWithCommand()`, `redeem()` or `redeemK()` of the `Index` contract, the function call will revert.

Recommendation: A possible fix would be to only grant the needed allowance before the external call, and after the external call has been performed, check that the remaining allowance is `0` to ensure no dangling approvals are left behind.

For example, the `CommandTarget` struct could be refactored to add an `allowanceAmount` field, which allows the caller to specify how much allowance to grant.

In `CommandLib._call()`, make the following change as described above:

```
- bool approve = allowanceTarget != address(0);
- if (approve) currency.approve(allowanceTarget, type(uint256).max);
+ if (allowanceTarget != address(0) && allowanceAmount != 0)
currency.approve(allowanceTarget, allowanceAmount);

  (success, returnData) = target.call{value: value}(data);
  if (!success) revert CallFailed(returnData);

- if (approve) currency.approve(allowanceTarget, 0);
+ if (currency.allowance(address(this), allowanceTarget) != 0) revert
DanglingApproval();
```

This should be applied to all instances of `approve(..., 0)` in the codebase.

Phuture: Acknowledged, we do not intend to support tokens that do not allow zero approvals.

Renascence: This finding was acknowledged.

[M-10] Using `uint96` to store balances is incompatible for tokens with low value or high decimals

Context:

- CurrencyLib.sol#L138-L139
- IndexCommandsLib.sol#L35

Description: Throughout the codebase, `uint96` is used to store token balances. An example of this would be using `a160u96` to store constituents, with the upper 96 bits for the token balance and the lower 160 bits for the token address.

However, `uint96` will be too small for tokens with high decimals or low value. `type(uint96).max` is around `7e28`, which can overflow in realistic scenarios.

For example, consider SHIB, which has 18 decimals and a current price of $0.000009714. To calculate how much USD worth of SHIB would overflow `uint96`:

```
type(uint96).max / 1e18 * 0.000009714 = 769622
```

As seen from above, an amount of SHIB worth more than $770K would overflow `uint96`. This is a problem in several parts of the codebase.

Firstly, when depositing reserve using `Index.deposit()`, `CurrencyLib.selfDeposit()` is called, which contains an unsafe cast of the amount transferred in to `uint96`:

```
uint256 _balance = token.balanceOf(address(this));
token.safeTransferFrom(msg.sender, address(this), amount);
// safe cast, transferred amount is <= 2^96-1
deposited = uint96(token.balanceOf(address(this)) - _balance);
```

If the reserve currency was SHIB and the user transferred in more than $770K worth of SHIB, `token.balanceOf(address(this)) - _balance` would be more than `uint96`. The unsafe cast would

then overflow, causing the user to lose nearly all of his deposited amount.

The same problem exists in `Index.depositWithCommand()`, since it uses an unsafe cast in `IndexCommandsLib.depositWithCommand()`:

```
deposited = uint96(reserve.balanceOfSelf() - reserveBefore);
```

Secondly, due to the use of `uint96` to represent balances throughout the codebase, such as in `OrderBook.sol` and `AnatomyValdiatonLib.sol`, the rebalancing process could revert if a constituent balance ends up becoming larger than `uint96`.

Recommendation: In `CurrencyLib.selfDeposit()` and `IndexCommandsLib.depositWithCommand()`, consider using `safeCastTo96()` instead of performing unsafe casts:

```
  // safe cast, transferred amount is <= 2^96-1
- deposited = uint96(token.balanceOf(address(this)) - _balance);
+ deposited = (token.balanceOf(address(this)) - _balance).safeCastTo96();
```

```
- deposited = uint96(reserve.balanceOfSelf() - reserveBefore);
+ deposited = (reserve.balanceOfSelf() - reserveBefore).safeCastTo96();
```

This ensures that `Index.deposit()` and `Index.depositWithCommand()` will revert if a user attempts to deposit more than `type(uint96).max` tokens.

Additionally, to ensure that rebalancing does not revert unexpectedly, consider not using tokens with high decimals or low prices as constituents.

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented. Importantly, the mitigation does not completely fix the issue and tokens that could lead to an overflow of `uint96` in the rebalancing process must be avoided.

[M-11] `msg.sender` **is not validated for** `ZeroExAdapter.phutureOnConsumeCallbackV1()`

Context:

- ZeroExAdapter.sol#L23-L36

Description: `ZeroExAdapter.phutureOnConsumeCallbackV1()` does not check if `msg.sender` is a whitelisted address. As such, anyone can call `phutureOnConsumeCallbackV1()` to perform arbitrary calls from `ZeroExAdapter` to any address:

```
(bool success,) = params.target.call{value: params.currencyIn.isNative() ?
params.amountIn : 0}(params.data);
if (!success) revert SwapFailed();
```

Although `ZeroExAdapter` does not hold any funds and has no state, this is still very dangerous. An attacker can use `phutureOnConsumeCallbackV1()` to manipulate the state of external contracts for `ZeroExAdapter`.

For example, to DOS rebalancing, an attacker can grant infinite USDT allowance from `ZeroExAdapter` to the 0x Exchange Proxy. Afterwards, when `phutureOnConsumeCallbackV1()` is called during rebalancing to swap USDT to other tokens, it will revert.

This is because USDT does not allow non-zero to non-zero approvals:

```
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

However, `phutureOnConsumeCallbackV1()` performs an non-zero approval before attempting a swap:

```
params.currencyIn.approve(params.target, params.amountIn);
```

As such, this call will always revert, making `ZeroExAdapter` permanently unusable for USDT swaps.

Recommendation: In `phutureOnConsumeCallbackV1()`, consider validating that `msg.sender` is the vault/index address:

```
  function phutureOnConsumeCallbackV1(bytes calldata data) external {
+     if (msg.sender != vault) revert NotAuthorized();
      TradeParams memory params = abi.decode(data, (TradeParams));
```

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented.

Low Risk

[L-1] `BaseIndex.setConfig()` should accrue AUM fee when `metadata` address is updated

Context:

- BaseIndex.sol#L96

Description: The `BaseIndex.setConfig()` function accrues the AUM fee up to `block.timestamp` whenever the `AUMDilutionPerSecond` changes as a result of the config change.

There is a check missing since when the `metadata` provider is updated, the AUM fee is not accrued even though it's likely that the `AUMDilutionPerSecond` has changed.

Recommendation:

```
## BaseIndex.sol

        if (
            _prevConfig.AUMDilutionPerSecond !=
_depositConfig.shared.AUMDilutionPerSecond
                || _prevConfig.useCustomAUMFee != _depositConfig.shared.useCustomAUMFee
+               || _prevConfig.metadata != _depositConfig.shared.metadata
        ) {
            uint96 AUMFee;
```

Note that even with this additional check it is implicitly assumed that the same `metadata` provider always returns the same `AUMDilutionPerSecond`.

Phuture: Fixed in PR 233

Renascence: The AUM fee is now accrued in all cases.

[L-2] `GovernanceProxy` does not restrict access to `receive()` function

Context:

- GovernanceProxy.sol#L7
- Proxy.sol#L75

Description: The OpenZeppelin `Proxy` contract that is used by the Phuture V2 protocol can be found here. It implements the `receive()` and `fallback()` function.

```
## Proxy.sol

/**
* @dev Fallback function that delegates calls to the address returned by
`_implementation()`. Will run if no other
* function in the contract matches the call data.
*/
fallback() external payable virtual {
    _fallback();
}


/**
* @dev Fallback function that delegates calls to the address returned by
`_implementation()`. Will run if call data
* is empty.
*/
receive() external payable virtual {
    _fallback();
}
```

On the other hand, the `GovernanceProxy` contract does not override the `receive()` function to protect it with an `onlyOwner` modifier.

This means that if the `Governance` contracts implement a function that can be called via the `receive()` function in the `Proxy`, it is unprotected and anyone can call it.

For example, if the `Governance` implements a `receive()` function itself, anyone can call it.

Currently, there is no risk since no privileged governance logic can be accessed by providing empty calldata.

Recommendation: It is recommended to override the `receive()` function in `GovernanceProxy` and to protect it with the `onlyOwner` modifier.

```
## GovernanceProxy.sol

          _delegate(_implementation());
      }

+     receive() external payable override onlyOwner {
+         _delegate(_implementation());
+     }
+
      function setImplementation(address implementation_) external onlyOwner {
          implementation = implementation_;
      }
```

Another option is to use a more recent OpenZeppelin version since newer versions don't implement the `receive()` function anymore.

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented.

[L-3] Hardcoded `sgReceive()` gas might not be sufficient for some chains

Context:

- OmnichainMessenger.sol#L48

Description: Although the hardcoded value should be enough to execute `sgReceive()` on most EVM chains, it is still encouraged to make this a configurable parameter.

Arbitrum has differences due to its specific precomiles: https://docs.arbitrum.io/arbos/gas

> The L2 component consists of the traditional fees Geth would pay to stakers in a vanilla L1 chain, such as the computation and storage charges applying the state transition function entails. ArbOS charges additional fees for executing its L2-specific precompiles, whose fees are dynamically priced according to the specific resources used while executing the call.

zkSync also has a different gas model: https://docs.zksync.io/build/developer-reference/fee-model.html#state-diffs-vs-transaction-inputs.

Taking a look at a simple `approve` transaction on zkSync we can observe it consumes ~665k gas: https://explorer.zksync.io/tx/0xcd69c39db39362136e3b347c17fdb756d600db0568d5154b516c86bde31f0a51

Using 500k gas for `sgReceive` on zkSync would certainly fail out-of-gas.

```
## OmnichainMessenger.sol

    function sgReceive(
        uint16 srcEid,
        bytes calldata srcAddress,
        uint256,
        address token,
        uint256 amountLD,
        bytes calldata payload
    ) external {
        if (msg.sender != stargateComposer) revert Forbidden();
        if (keccak256(abi.encodePacked(srcAddress, address(this))) !=
keccak256(trustedRemotes[srcEid])) return;

        Currency currency = token == sgETH ? CurrencyLib.NATIVE :
Currency.wrap(token);

        (bytes32 ordersHash, uint256 chainId) = abi.decode(payload, (bytes32,
uint256));
        ordersHashOf[chainId] = keccak256(abi.encode(ordersHash, currency, amountLD));

        currency.transfer(address(orderBook), amountLD);
    }
```

Other EVM chains that should be supported also might have different gas models.

Recommendation: The maximum gas spent for `sgReceive()` should be estimated per chain and configured properly.

```
## OmnichainMessenger.sol

abstract contract OmnichainMessenger is BlockingApp, IStargateReceiver, IOmnicha
         uint8 decimals;
     }

+    error SgReceiveGasNotSet();

-    uint256 internal constant SG_RECEIVE_GAS = 500_000;
+    mapping(uint16 => uint256) internal sgReceiveGas;

             IStargateRouter.lzTxObj memory lzTxObj;
-            lzTxObj.dstGasForCall = SG_RECEIVE_GAS;
+            lzTxObj.dstGasForCall = getSgReceiveGas(dstEid);
             (uint256 sgMessageFee,) =
                 IStargateRouter(stargateComposer).quoteLayerZeroFee(dstEid, 1, to,
payload, lzTxObj);

+    function setSgReceiveGas(uint16 eid, uint256 gas) external onlyOwner {
+        sgReceiveGas[eid] = gas;
+    }
+
+    function getSgReceiveGas(uint16 eid) internal view returns (uint256) {
+        uint256 gas = sgReceiveGas[eid];
+        if (gas == 0) revert SgReceiveGasNotSet();
+        return gas;
+    }
+
```

Phuture: Fixed in PR 230

Renascence: The `sgParams` struct that specifies the gas to execute `sgReceive()` with on the destination chain is now dynamically passed in by the governance for every function call. The hardcoded value has been removed.

[L-4] `ChainRegistryLib` should limit `index` to 254

Context:

- ChainRegistryLib.sol#L33
- ConfigBuilder.sol#L38

Description: The `ConfigBuilder` has the `MAX_CHAIN_INDEX_COUNT = 255` constant which aims to limit the number of chains that can be registered to `255`.

This is necessary since the number of chains is cast to `uint8` in two instances in the `ConfigBuilder` and any number > `255` would result in an overflow.

Creating a `BitSet` with `BitSet.create(255)` does not restrict the number of chains to `255` since the `BitSet` will consist of one word which can hold `256` chain indexes.

The restriction to `255` chains must therefore be applied in the `ChainRegistryLib.register()` function.

Recommendation:

```
## ChainRegistryLib.sol

     if (packedIndex != 0) revert AlreadyRegistered(_unpackIndex(packedIndex));

     index = self.indexSet.size();
+    require(index < 255, "too many chains");
     self.indexSet = self.indexSet.add(index);
     self.chainsHash = keccak256(abi.encode(self.chainsHash, chainId, index));
```

By restricting the chain indexes to `[0,254]` it is ensured that only `255` chains can be registered.

Phuture: Fixed in PR 223

Renascence: The recommendation has been implemented.

[L-5] Rounding in `OmnichainMessenger.pushIncomingOrders()` prevents removing currencies from anatomy

Context:

- OmnichainMessenger.sol#L136
- AnatomyValidationLib.sol#L66

Description: `AnatomyValidationLib.validate()` does not allow removing a currency from the anatomy if it has a `balance > 0`.

```
## AnatomyValidationLib.sol

    if (balance == 0) {
        // new anatomy shouldn't contain any currencies without balances
        if (params.newAnatomy.currencyIndexSet.contains(i)) revert
ExcessNewAnatomyCurrencyIndex(i);
    } else {
        // new anatomy must contain all non-zero currency balances
>       if (!params.newAnatomy.currencyIndexSet.contains(i)) revert
NewAnatomyCurrencyIndexNotFound(i);
```

The issue is that `OmnichainMessenger.pushIncomingOrders()` rounds down the sell amounts for the incoming orders.

As a result, the sum of the `sellAmount` values doesn't add up to `receivedAmount` and the affected currency MUST be part of the new anatomy.

There is no immediate security threat but the anatomy can become bloated with unused currencies. The small dust amounts that are added to the anatomy however lead to downstream problems as described in other findings that discuss issues for small amounts in detail.

Recommendation: The sum of the sell amounts in `OmnichainMessenger.pushIncomingOrders()` must be ensured to add up to `receivedAmount`.

```
## OmnichainMessenger.sol

         OrderLib.Order[] memory orders = new OrderLib.Order[](boughtOrders.length);
+        uint256 lastIndex = boughtOrders.length - 1;
+        uint256 utilized;
+        uint256 sellAmount;
         for (uint256 i; i < boughtOrders.length; i++) {
+            sellAmount = (i == lastIndex) ?
+                    (receivedAmount - utilized):
+                    boughtOrders[i].amount.mulDivDown(receivedAmount,
boughtOrdersTotalAmount);
             orders[i] = OrderLib.Order({
-                sellAmount: boughtOrders[i].amount.mulDivDown(receivedAmount,
boughtOrdersTotalAmount).safeCastTo96(),
+                sellAmount: sellAmount.safeCastTo96(),
                 idParams: OrderLib.OrderId({
                     sellCurrency: currency,
                     localBuyCurrency: boughtOrders[i].buyCurrency,
@@ -141,6 +147,7 @@ abstract contract OmnichainMessenger is BlockingApp,
IStargateReceiver, IOmnicha
                     finalDestinationChainId: block.chainid
                 })
             });
+            utilized += sellAmount;
         }

         orderBook.receiveIncomingOrders(orders, currency, receivedAmount);
```

Phuture: Fixed in PR 223

Renascence: The recommendation has been implemented.

[L-6] `OrderBook.finishOrderExecution()` must not return empty `BoughtOrders`

Context:

- OrderBook.sol#L150-L201

Description: When the `OrderBook.finishOrderExecution()` function is left it should be ensured that for every `chainIndex` there are no empty orders in `pendingOrders[chainIndex].orders` as otherwise empty orders are sent to the destination chain and `incomingOrders -= int256(orders.length);` reverts.

Recommendation: The fix checks that `pendingOrders[chainIndex].orders.length == orderIndex` when the logic moves from one chain to the next. A separate check is needed for the last `chainIndex`.

```
## OrderBook.sol

                if (orderId.finalDestinationChainId != lastRemoteChainId) {
                    if (lastRemoteChainId != type(uint256).max) {
                        if (orderId.finalDestinationChainId < lastRemoteChainId)
revert Sorting();
+                       require(pendingOrders[chainIndex].orders.length ==
orderIndex);

                        orderIndex = 0;
                        unchecked {
@@ -198,6 +199,9 @@ contract OrderBook is IOrderBook, IPhutureOnDonationCallback,
Owned {
                }
            }
        }
+       if (pendingOrders.length > 0) {
+           require(pendingOrders[chainIndex].orders.length == orderIndex);
+       }
    }

    function phutureOnDonationCallbackV1(bytes memory data) external override
only(vault) {
```

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented.

[L-7] Calculation of `orderSellAmount` in `RebalancingLib.previewReserveRebalancingOrders()` should round down

Context:

- RebalancingLib.sol#L199-L201

Description: In `RebalancingLib.previewReserveRebalancingOrders()`, when splitting the reserve balance into the `orderSellAmount` for each constituent in the index, the calculation rounds up:

```
## RebalancingLib.sol

uint96 orderSellAmount = vars.weightIndex == vars.lastIndex
    ? reserveAmount - vars.utilized
    : reserveAmount.mulDivUp(weights[vars.weightIndex], MAX_WEIGHT).safeCastTo96();
```

However, the calculation for `orderSellAmount` here should round down instead, otherwise it's theoretically possible for `vars.utilized` to be larger than `reserveAmount` for the last currency's calculation.

A naive example with simple numbers:

- `MAX_WEIGHT = 10`
- `weights = [3, 3, 3, 1]`
- `reserveAmount = 4`

- For each weight before `1`, `4 * 3 / 10` rounds up to `2`.
- So at the last iteration where `weight = 1`, we have `vars.utilized = 6` but `reserveAmount = 4`, causing the calculation to revert.

This is likely to occur when the last currency is allocated an extremely small portion of `reserveAmount`, such that the error from rounding is larger than the allocated amount for the last currency. If the reserve currency is a token with low decimals, this is more likely to occur.

This would cause `Configbuilder.startReserveRebalancing()` to revert when attempting to rebalance an index's reserves.

Recommendation: Use `mulDivDown()` instead of `mulDivUp()`:

```
## RebalancingLib.sol

  uint96 orderSellAmount = vars.weightIndex == vars.lastIndex
      ? reserveAmount - vars.utilized
-     : reserveAmount.mulDivUp(weights[vars.weightIndex], MAX_WEIGHT).safeCastTo96();
+     : uint96(reserveAmount.mulDivDown(weights[vars.weightIndex], MAX_WEIGHT));
```

Note that `.safeCastTo96()` isn't used here since there is no need to.

`reserveAmount` is a `uint96` and `weights[vars.weightIndex]` is guaranteed to be smaller than or equal to `MAX_WEIGHT`, so the result will never exceed `uint96`.

Phuture: Fixed in PR 239

Renascence: The recommendation has been implemented.

[L-8] Price calculation in `ChainlinkLib` cannot handle tokens with large decimals

Context:

- ChainlinkLib.sol#L25-L28

Description: In `ChainlinkLib.sol`, the Q128-scaled price for an asset is calculated as such:

```
## ChainlinkLib.sol

price_ = PriceLib.Q128.mulDivDown(
    10 ** (aggregators[0].aggregatorDecimals + decimals),
    _getChainlinkPrice(aggregators[0].aggregatorAddress, staleAfter) *
PriceLib.DECIMALS_MULTIPLIER
);
```

Where:

- `aggregators[0].aggregatorDecimals` is the number of decimals for the Chainlink price feed
- `decimals` is the token's decimals
- `PriceLib.DECIMALS_MULTIPLIER` is `1e18`

However, if `aggregators[0].aggregatorDecimals + decimals` is larger than 38, `PriceLib.Q128` multiplied by `10 ** (aggregators[0].aggregatorDecimals + decimals)` will be greater than `type(uint256).max`, causing `mulDivDown()` to revert.

For example, if the token had 21 decimals, and its Chainlink price feed had 18 decimals:

```
aggregatorDecimals = 21
decimals = 18
PriceLib.Q128 * (10 ** (aggregatorDecimals + decimals)) = (2 ** 128) * 1e39 = 3.4e77
```

Since `3.4e77` is larger than `type(uint256).max`, the calculation reverts due to an arithmetic overflow.

Recommendation: Ensure that `ChainlinkPriceOracle` is never configured such that the sum of the aggregator and token's decimals is more than 38.

Phuture: Acknowledged. `PriceOracle` will be updated If we need to support tokens with 21+ decimals in the future. This change was not implemented as of now since it might lead to a decrease in price accuracy.

Renascence: The finding has been acknowledged.

[L-9] Mapped transfers in `CommandLib._executeCommand()` could silently fail

Context:

- CommandLib.sol#L66-L76
- CommandLib.sol#L102-L103

Description: In `CommandLib._executeCommand()`, when a command's `mapTarget` flag is set to `true`, the command will be mapped to either a native transfer or an ERC-20 transfer:

```
## CommandLib.sol

if (cmd.mapTarget) {
    Info memory targetInfo = validator.mapTarget(
        currencyStates[currencyIndex], cmd.target, pc & PACKED_CONFIG_MASK,
cmd.datas[k]
    );
    _call(
        currencyStates[currencyIndex].currency,
        targetInfo.target,
        targetInfo.allowanceTarget,
        targetInfo.value,
        targetInfo.data
    );
```

As seen from above, the transfer is then performed with `_call()`.

However, since `_call()` only checks if `success == true` but doesn't check `returndata`, commands that are mapped to ERC-20 transfers will not check the return value `transfer()`:

```
## CommandLib.sol

(success, returnData) = target.call{value: value}(data);
if (!success) revert CallFailed(returnData);
```

This could cause transfers for no-revert-on-failure tokens to silently fail, such as EURS:

```
function transfer (address _to, uint256 _value) public payable returns (bool success)
{
  uint256 fromBalance = accounts [msg.sender];
  if (fromBalance < _value) return false;
```

Recommendation: Check `returndata` for commands mapped to ERC20 transfers as such:

```
## CommandLib.sol

- _call(
+ (, bytes memory returndata) = _call(
      currencyStates[currencyIndex].currency,
      targetInfo.target,
      targetInfo.allowanceTarget,
      targetInfo.value,
      targetInfo.data
  );
+ if (bytes4(targetInfo.data) == IERC20.transfer.selector && returndata.length != 0
&& !abi.decode(returndata, (bool))) revert TransferFailed();
```

Phuture: Acknowledged. We do not intend to support no-revert-on-failure tokens.

Renascence: The finding has been acknowledged.

[L-10] Dont reset `rebalancingFlags` when `BaseIndex.setConfig()` is called outside the context of `HomechainGovernance.finishRebalancing()`

Context:

- BaseIndex.sol#L120
- HomechainGovernance.sol#L48
- HomechainGovernance.sol#L78

Description: The `BaseIndex.setConfig()` function is called from three functions in the `HomechainGovernance`:

- `HomechainGovernance.setInitialConfig()`
- `HomechainGovernance.setConfig()`
- `HomechainGovernance.finishRebalancing()`

If rebalancing is not finished it is possible that the `governance` by mistake calls `HomechainGovernance.setConfig()`, thereby resetting the `rebalancingFlags` flag and allowing users to deposit

funds into the index / withdraw funds from the index. This is not safe and leaves the protocol in an inconsistent state.

Therefore, `governance` should be protected from performing this erroneous action.

Recommendation: The `BaseIndex.setConfig()` function should take a `clearRebalancingFlags` parameter that indicates if the function is called from `HomechainGovernance.finishRebalancing()`.

```
## BaseIndex.sol

     function setConfig(
         Config calldata _prevConfig,
         DepositConfig calldata _depositConfig,
-        RedemptionConfig calldata _redemptionConfig
+        RedemptionConfig calldata _redemptionConfig,
+        bool clearRebalancingFlags
     ) external onlyOwner {
         if (keccak256(abi.encode(_depositConfig.shared)) !=
keccak256(abi.encode(_redemptionConfig.shared))) {
             revert IndexConfigMismatch();
@@ -117,7 +118,7 @@ abstract contract BaseIndex is Vault, IBaseIndex, IERC20Metadata {
         depositConfigHash = keccak256(abi.encode(_depositConfig));
         redemptionConfigHash = keccak256(abi.encode(_redemptionConfig));

-        state.rebalancingFlags &= ~uint8(0x10);
+        if (clearRebalancingFlags) state.rebalancingFlags &= ~uint8(0x10);

         emit ConfigUpdated(encodedConfig);
     }


## HomechainGovernance.sol

         (IBaseIndex.DepositConfig memory depositConfig, IBaseIndex.RedemptionConfig
memory redemptionConfig) =
             IConfigBuilder(configBuilder).setConfig(baseConfig);
         IBaseIndex.Config memory config;
-        IBaseIndex(index).setConfig(config, depositConfig, redemptionConfig);
+        IBaseIndex(index).setConfig(config, depositConfig, redemptionConfig, false);

         emit SetConfigBuilder(configBuilder);
     }
@@ -45,7 +45,7 @@ contract HomechainGovernance is Governance {

         (IBaseIndex.DepositConfig memory depositConfig, IBaseIndex.RedemptionConfig
memory redemptionConfig) =
             IConfigBuilder(configBuilder).setConfig(baseConfig);
-        IBaseIndex(index).setConfig(currentConfig, depositConfig, redemptionConfig);
+        IBaseIndex(index).setConfig(currentConfig, depositConfig, redemptionConfig,
false);
     }

     function startRebalancing(
@@ -75,7 +75,7 @@ contract HomechainGovernance is Governance {
     ) external {
         (IBaseIndex.DepositConfig memory deposit, IBaseIndex.RedemptionConfig memory
redemption) =
             IConfigBuilder(configBuilder).finishRebalancing(results, config);
-        IBaseIndex(index).setConfig(currentConfig, deposit, redemption);
```

```
+         IBaseIndex(index).setConfig(currentConfig, deposit, redemption, true);
          IHomechainOmnichainMessenger(messenger).setLayerZeroConfig(lzConfig);
      }
```

Phuture: Fixed in PR 232 and PR243.

Renascence: The recommended solution has not been implemented. Instead, the `HomechainGovernance.setConfig()` function no longer calls `BaseIndex.setConfig()`. `BaseIndex.setConfig()` is only called as part of finishing rebalancing (and when the initial config is set).

Furthermore, in PR243, `ConfigBuilder.setConfig()` can also only be called in the rebalancing phase, keeping the configuration in `ConfigBuilder` and `BaseIndex` in sync.

Informational

[I-01] `Governance` **is missing functions to access** `setSequencerUptimeFeed()` **and** `withdrawCurrency()`

Context:

- L2Index.sol#L86
- OmnichainMessenger.sol#L96

Description: The governance contracts don't implement functions to call `OmnichainMessenger.withdrawCurrency()` or `L2Index.setSequencerUptimeFeed()`.

It is possible to upgrade the governance contracts and implement the functions but it's better to implement the functions now to have them immediately accessible when needed.

Recommendation: `L2Index.setSequencerUptimeFeed()` is only accessible on the homechain, so it should be implemented in `HomechainGovernance`.

```
## HomechainGovernance.sol

     function accrueFee(address index, address recipient) external {
         IBaseIndex(index).accrueFee(recipient);
     }
+
+    function setSequencerUptimeFeed(address index, address sequencerUptimeFeed)
external {
+        L2Index(index).setSequencerUptimeFeed(sequencerUptimeFeed);
+    }
 }
```

`OmnichainMessenger.withdrawCurrency()` must be accessible on the homechain and the remote chains, so it should be implemented in `Governance`.

```
## Governance.sol

         IOmnichainMessenger(messenger).setHomeEid(homeEid);
     }

+    function withdrawCurrency(address messenger, Currency currency, address to,
uint256 amount) external {
+        IOmnichainMessenger(messenger).withdrawCurrency(currency, to, amount);
+    }
+
     function transferContractOwnership(address owned, address newOwner) external {
         Owned(owned).transferOwnership(newOwner);
     }
```

Phuture: Fixed in PR 222

Renascence: The recommendation has been implemented. However, the `priceOracleDeployer` variable in `HomechainGovernance.setSequencerUptimeFeed()` should be renamed to `index`.

[I-02] Remove outer loop in `ConfigBuilder.getState()`

Context:

- ConfigBuilder.sol#L372

Description: `ConfigBuilder.getState()` iterates over all words in the `chainIdSet`. Since this set always contains 1 word, the word can be accessed directly.

Recommendation:

```
## ConfigBuilder.sol

@@ -369,17 +379,15 @@ contract ConfigBuilder is IConfigBuilder, IConfigMigration {
         state.hasRemote = hasRemote;

         uint256 chainIndex;
-        for (uint256 wi; wi < state.chainIdSet.length; ++wi) {
-            uint256 w = state.chainIdSet[wi];
-            for (uint256 b = BitSet.find(w, 0); BitSet.hasNext(w, b);) {
-                uint256 id = BitSet.valueAt(wi, b);
+        uint256 w = state.chainIdSet[0];
+        for (uint256 b = BitSet.find(w, 0); BitSet.hasNext(w, b);) {

-                state.chainStates[chainIndex] = ChainState(currenciesHashOf[id],
resultHashOf[id]);
+            state.chainStates[chainIndex] = ChainState(currenciesHashOf[b],
resultHashOf[b]);

-                unchecked {
-                    ++chainIndex;
-                    b = BitSet.find(w, b + 1);
-                }
+            unchecked {
+                ++chainIndex;
+                b = BitSet.find(w, b + 1);
             }
         }
     }
```

Phuture: Fixed in PR 231

Renascence: The recommendation has been implemented.

[I-03] Custom configuration for LayerZero

Context:

- BlockingApp.sol#L28

Description: Some chains like Polygon have deep reorg issues: https://protos.com/polygon-hit-by-157-block-reorg-despite-hard-fork-to-reduce-reorgs/.

Recommendation: The default LayerZero configuration should be sufficient not to be affected by chain reorgs: https://layerzero.gitbook.io/docs/technical-reference/mainnet/default-config.

It is advised being conservative when setting custom `CONFIG_TYPE_OUTBOUND_BLOCK_CONFIRMATIONS` and `CONFIG_TYPE_INBOUND_BLOCK_CONFIRMATIONS` for the `BlockingApp`.

Phuture: Acknowledged.

Renascence: The recommendation has been acknowledged.

[I-04] LayerZero payload size limit

Context:

- BlockingApp.sol#L81

Description: LayerZero added an explicit check for the maximum size payload in their current relayer implementation: https://github.com/LayerZero-Labs/LayerZero/commit/7d07e87db7f813e2c233142e55639ae797bb65fb.

This is the Relayer for the `UltraLightNodeV2` which is the current default library.

If they introduce a new default library/relayer and this check is not added it would be possible to send bigger payloads.

This can lead to the following issue: https://solodit.xyz/issues/h-2-malicious-user-can-use-an-excessively-large-_toaddress-in-oftcoresendfrom-to-break-layerzero-communication-sherlock-uxd-uxd-protocol-git.

The likelihood of the new default relayer not having this check is extremely low so this is not a security risk.

Recommendation: It is still advised to add an explicit check for max payload size inside the `BlockingApp`, the same way it's done in the LzApp example contract.

```
## BlockingApp.sol

@@ -13,6 +13,12 @@ abstract contract BlockingApp is IBlockingApp, Owned {
     mapping(uint16 => bytes) public trustedRemotes;
     mapping(uint256 => PoolIds) public poolIds;

+    uint public constant DEFAULT_PAYLOAD_SIZE_LIMIT = 10000;
+
+    mapping(uint16 => uint) public payloadSizeLimitLookup;
+
+    error PayloadSizeLimitExceeded(uint16 dstEid, uint payloadSize);
+
     event TrustedRemoteSet(uint16 remoteEid, bytes path, PoolIds poolIds);
```

```
     error OnlyEndpoint();
@@ -67,6 +73,22 @@ abstract contract BlockingApp is IBlockingApp, Owned {
         return lzEndpoint.getConfig(version, eid, address(this), configType);
     }

+    function _checkPayloadSize(uint16 dstEid, uint _payloadSize) internal view
virtual {
+        uint payloadSizeLimit = payloadSizeLimitLookup[dstEid];
+        if (payloadSizeLimit == 0) {
+            // use default if not set
+            payloadSizeLimit = DEFAULT_PAYLOAD_SIZE_LIMIT;
+        }
+        if (_payloadSize > payloadSizeLimit) {
+            revert PayloadSizeLimitExceeded(dstEid, _payloadSize);
+        }
+    }
+
+    // if the size is 0, it means default size limit
+    function setPayloadSizeLimit(uint16 _dstChainId, uint _size) external onlyOwner {
+        payloadSizeLimitLookup[_dstChainId] = _size;
+    }
+
     function _lzSend(
         uint16 dstEid,
         bytes memory path,
@@ -77,6 +99,8 @@ abstract contract BlockingApp is IBlockingApp, Owned {
     ) internal {
         if (zroPaymentAddress != address(0)) revert ZRONotSupported();

+        _checkPayloadSize(dstEid, message.length);
+
         // solhint-disable-next-line check-send-result
         lzEndpoint.send{value: address(this).balance}(dstEid, path, message,
refundAddress, address(0), options);
     }
```

Phuture: Acknowledged.

Renascence: The finding has been acknowledged. LayerZero must continue implementing this check.

[I-05] Remove `index` from chains hash and currencies hash calculations

Context:

- ChainRegistryLib.sol#L35
- CurrencyRegistryLib.sol#L35

Description: The `chainsHash` and `currenciesHash` are calculated such that chains and currencies arrays can be passed to the functions instead of storing the data on-chain and reading it from storage.

It must be possible to verify that all chains and currencies are provided and that they are ordered correctly.

Currently, the hashes are calculated in this way: `hash[i+1] = keccak256(abi.encode(hash[i],data,index))`.

Presumably, the `index` is part of the hash to ensure the correct ordering. But the ordering is ensured even without the `index` as it's not possible to arrive at the same final hash by ordering the data (currency or chain) in any other way.

Recommendation: The `index` can be removed from the hash calculations.

```
## ChainRegistryLib.sol

          index = self.indexSet.size();
          self.indexSet = self.indexSet.add(index);
-         self.chainsHash = keccak256(abi.encode(self.chainsHash, chainId, index));
+         self.chainsHash = keccak256(abi.encode(self.chainsHash, chainId));

          self.packedIndexOf[chainId] = index + 1; // implicit overflow check
      }


## CurrencyRegistryLib.sol

          // register new address
          index = self.currencies.length;
          self.currencies.push(currency);
-         newHash = keccak256(abi.encode(self.currenciesHash, currency, index));
+         newHash = keccak256(abi.encode(self.currenciesHash, currency));
          self.currenciesHash = newHash;
          self.packedIndexOf[currency] = _packIndex(index);
      }
```

Note that in all instances where the hash is calculated to verify it, the `index` needs to be removed as well.

Phuture: Fixed in PR 218

Renascence: The recommendation has been implemented.

[I-06] `OmnichainMessenger` does not work on Aptos

Context:

- OmnichainMessenger.sol#L157

Description: Some non-EVM chains like Aptos have addresses bigger than 20 bytes. Expanding the index to those chains and casting the address to `bytes20` doesn't work.

Recommendation: If these chains should be supported, a registry of address sizes for different chains should be kept. Using the registry, the address can be retrieved from the path.

Phuture: Acknowledged. We do not intend to support non-EVM compatible chains, including Aptos, in the current version of the contracts.

Renascence: The finding has been acknowledged.

[I-07] Lack of validation for Stargate chain paths

Context:

- OmnichainMessenger.sol#L172

Description: The Phuture team wants to support the following chains:

- Manta Pacific
- Arbitrum
- Optimism + (OP Superchain = Base and Mode)
- Mantle
- Metis
- zkSync
- Linea
- Avalanche
- Ethereum

By taking a look at chains supported by Stargate: `https://stargateprotocol.gitbook.io/stargate/developers/pool-ids`, a few that aren't supported can be eliminated, e.g. zkSync, MantaPacific.

What is more subtle is what pools a specific chain supports. Most of the chains support some stablecoin. The exception is Linea, which only supports ETH.

Here is the list of connected token paths across different chains: `https://stargateprotocol.gitbook.io/stargate/developers/stargate-chain-paths`.

For example, a bad anatomy would be creating an index with Linea and Kava. They couldn't transfer funds between each other with Stargate as they don't have any common chain path.

Recommendation: As a general rule, any new chain that is added needs to have a common asset with all the other chains in the index.

These checks are not performed on-chain, so the governance is expected to analyze any new chain before it's added.

Phuture: Acknowledged.

Renascence: The information has been acknowledged.

[I-08] Improve efficiency of trading algorithm in `RebalancingLib`

Context:

- RebalancingLib.sol#L144-L145

Description: The trading algorithm is described by the following pseudocode.

```
for chain in chains:
    for currency in currencies[chain]:
        if (currentInBase < targetInBase):
            // add buy order for targetInBase - currentInBase to stack
        else if (currentInBase > targetInBase):
            // add sell order for currentInBase - targetInBase to stack
        // match orders on stack until sell side or buy side is exhausted
```

It is useful to consider the outcome of the algorithm to be a directed graph where vertices are the chains and edges are the liquidity flows.

The algorithm has the following properties:

- A chain can have both incoming liquidity and outgoing liquidity at the same time. This incurs unnecessary slippage for cross-chain trades.
- For `n` chains it is possible that all `n` chains depend on each other for liquidity. By reducing the maximum "depth", the algorithm can be sped up.
- The graph is acyclic. This means the chains cannot get stuck in a deadlock. This is an important observation and it also holds true after the recommendation is applied.

By matching orders only after all currencies for a chain have been processed, the following behavior can be achieved:

- A chain cannot have both incoming and outgoing liquidity. This minimizes the cross-chain trading volume.
- The maximum "depth" is `2`.

Recommendation: The improvements can simply be implemented by calling `RebalancingLib._createOrders()` after all currencies for a chain have been iterated over, as opposed to calling it for each currency.

```
## RebalancingLib.sol

                withdrawal.currencyIndexSet.add(vars.j);
            }

-           (vars.sellDeltas, vars.buyDeltas) =
-               _createOrders(chainOrders, vars.sellDeltas, vars.buyDeltas,
vars.counters);
-       unchecked {
                if (vars.priorAsset) ++vars.currencyIndex;
                ++vars.j;
            }
        }

+       (vars.sellDeltas, vars.buyDeltas) =
+               _createOrders(chainOrders, vars.sellDeltas, vars.buyDeltas,
vars.counters);
+       if (vars.currenciesHash != currenciesHashOf[vars.i]) {
            revert CurrenciesHashMismatch(params.chainIds[vars.i]);
        }
```

Phuture: Fixed in PR 218

Renascence: The recommendation has been implemented.

[I-09] `CurrencyLib` should implement code from latest solmate version

Context:

- CurrencyLib.sol#L61-L84

Description: The `CurrencyLib.transfer()` function references the solmate implementation. However, the referenced code is from 2 years ago.

Since then, some changes have been applied. They are not needed from a security perspective but still recommended. The reasoning for the changes can be found in the solmate changelog.

Recommendation: Copy the code for the ERC20 transfer from the latest solmate implementation of the `safeTransfer()` function.

Phuture: Fixed in PR 222

Renascence: The recommendation has been implemented.

[I-10] `WORD_INDEX_MAX` constant in `BitSet` is redundant and can be removed

Context:

- BitSet.sol#L7
- BitSet.sol#L86

Description: Presumably, the `WORD_INDEX_MAX` constant has been used in a prior version of the `BitSet` library when the number of words has been passed to `BitSet.create()`.

Now the `BitSet.create()` function takes the number of values that the `BitSet` should be able to hold as the parameter. There doesn't have to be a limit since `create()` overflows and reverts when `maxSize >= 2^256 - type(uint8).max`.

Recommendation: The `WORD_INDEX_MAX` variable should be removed along with the check in `BitSet.create()`.

```
## BitSet.sol

 library BitSet {
     uint256 private constant WORD_SHIFT = 8;
-    // max word index is x * 256 + 255 = 2^256 -> x = (2^256 - 1 - 254) / 256
-    uint256 private constant WORD_INDEX_MAX = (type(uint256).max - 254) » WORD_SHIFT;
-
-    error Overflow();

     function hasNext(uint256 word, uint256 bit) internal pure returns (bool r) {
         assembly ("memory-safe") {
@@ -83,7 +79,6 @@ library BitSet {
     }

     function create(uint256 maxSize) internal pure returns (uint256[] memory bitset)
{
-        if (maxSize >= WORD_INDEX_MAX) revert Overflow();
         bitset = new uint256[](_capacity(maxSize));
     }
```

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented.

[I-11] `AnatomyValidationLib.validate()` should validate size of `currencyIndexSet`

Context:

- AnatomyValidationLib.sol#L29

Description: The `AnatomyValidationLib.validate()` function should check that all currencies in `params.currencyIndexSet` have been considered. This ensures that the set does not contain any currencies that have not been registered yet. Since the `params.currencyIndexSet` parameter is provided by the `governance` which is fully trusted, there is no security impact.

Still, it is recommended to implement the validation to prevent `governance` mistakes.

Recommendation:

```
## AnatomyValidationLib.sol

     error ExcessNewAnatomyCurrencyIndex(uint256 currencyIndex);
     error NewAnatomyCurrencyIndexNotFound(uint256 currencyIndex);
     error NewAnatomyCurrencyCountMismatch(uint256 expectedCount);
+    error NewAnatomyCurrencyIndexSetSizeMismatch(uint256 expectedCount);
     error NewAnatomyCurrencyMismatch(a160u96 expectedCurrency);

     function validate(
@@ -82,5 +83,9 @@ library AnatomyValidationLib {
         if (indexes.newAnatomy != params.newAnatomy.currencies.length) {
             revert NewAnatomyCurrencyCountMismatch(indexes.newAnatomy);
         }
+
+        if (indexes.newAnatomy != params.newAnatomy.currencyIndexSet.size()) {
+            revert NewAnatomyCurrencyIndexSetSizeMismatch(indexes.newAnatomy);
+        }
     }
 }
```

Phuture: Fixed in PR 231

Renascence: The recommendation has been implemented.

[I-12] Missing validation of orders hash in `RemoteOmnichainMessenger.finishRebalancing()`

Context:

- RemoteOmnichainMessenger.sol#L84

Description: The `RemoteOmnichainMessenger.finishRebalancing()` function should check that `ordersHashOf[block.chainid] == bytes32(0)`, as otherwise the `governance` may accidentally call `RemoteOmnichainMessenger.finishRebalancing()` without anyone having called `RemoteOmnichainMessenger.pushOrders()` before.

Since the `governance` is fully trusted, there is no security impact to this finding.

Recommendation:

```
## RemoteOmnichainMessenger.sol

         address zroPaymentAddress,
         bytes calldata options
     ) external payable onlyOwner {
+        require(ordersHashOf[block.chainid] == bytes32(0));
         // gather pending orders from order book and distribute them across the
chains using Stargate
         _distributeOrders(orderBook.finishOrderExecution(orderBookParams));
```

Phuture: Fixed in PR 231

Renascence: The recommendation has been implemented.

[I-13] Missing validation of `params.pendingOrderCounts.length` in `OrderBook.finishOrderExecution()`

Context:

- OrderBook.sol#L150-L201

Description: `OrderBook.finishOrderExecution()` does not validate that `params.pendingOrderCounts` has the correct length. This can lead to empty elements in the returned `PendingOrder[]` array.

There is no issue since empty `PendingOrder` structs lead to a revert in the downstream `OmnichainMessenger._distributeOrders()` function since `0` isn't a valid `block.chainId`.

```
## OmnichainMessenger.sol

uint16 dstEid = eIds[pendingOrder.chainId];
PoolIds memory sgPoolIds = poolIds[pendingOrder.chainId];

bytes memory to = bytes.concat(bytes20(_getPathOrRevert(dstEid)));
```

Recommendation:

```
## OrderBook.sol

                 }
             }
         }
+        require(params.pendingOrderCounts.length <= chainIndex + 1);
     }

     function phutureOnDonationCallbackV1(bytes memory data) external override
only(vault) {
```

Note that the <= accounts for the case when both `params.pendingOrderCounts.length` and `chainIndex` are equal to zero.

Phuture: Fixed in PR 233

Renascence: The recommendation has been implemented.

[I-14] `evm_version` is misspelt in `foundry.toml`

Context:

- foundry.toml#L5

Description: In the project's `foundry.toml`, `evm_version` is misspelt:

```
## foundry.toml

emv_version = "paris"
```

This is especially relevant since the protocol is intended to be deployed on many different chains, some of which might still not have support for the `PUSH0` opcode.

Fortunately, Foundry's default `evm_version` is currently also `paris`. Nevertheless, this should be corrected.

Recommendation:

```
## foundry.toml

- emv_version = "paris"
+ evm_version = "paris"
```

Phuture: Fixed in PR 218

Renascence: The recommendation has been implemented.

[I-15] Logic in `IndexCommandsLib.executeCommands()` can be simplified

Context:

- IndexCommandsLib.sol#L54-L58

Description: The following logic:

```
## IndexCommandsLib.sol

for (uint256 i; i < commandParams.additionalCurrencies.length; ++i) {
    balanceStates[i + length - commandParams.additionalCurrencies.length] =
CommandLib.BalanceState(
        commandParams.additionalCurrencies[i],
commandParams.additionalCurrencies[i].balanceOfSelf()
    );
}
```

can be simplified as such:

```
## IndexCommandsLib.sol

  for (uint256 i; i < commandParams.additionalCurrencies.length; ++i) {
-     balanceStates[i + length - commandParams.additionalCurrencies.length] =
CommandLib.BalanceState(
+     balanceStates[currencies.length + i] = CommandLib.BalanceState(
          commandParams.additionalCurrencies[i],
commandParams.additionalCurrencies[i].balanceOfSelf()
      );
  }
```

Phuture: Fixed in PR 223

Renascence: The recommendation has been implemented.

[I-16] Use separate `additionalGas` values for each callback to allow for fine-grained gas control

Context:

- HomechainOmnichainMessenger.sol#L249-L279

Description: The `HomechainOmnichainMessenger._sendSnapshotTransferAndTrades()` function iterates over `sendParams.batches` and the user is able to specify the `additionalGas` value per batch.

```
## IHomechainOmnichainMessenger.sol

struct Batches {
    address escrow;
    bytes[] callbacks;
    uint256 additionalGas;
}
```

The problem is that each batch can contain multiple `callbacks` that have different gas requirements, and to make all `callbacks` work, the `additionalGas` must be set to the maximum gas consumption of all `callbacks`.

As a result, the user needs to overpay for gas.

Recommendation: It is recommended to have one `additionalGas` value per `callback`. The `Batches` struct can simply be changed to have an `additionalGas` array.

```
## IHomechainOmnichainMessenger.sol

  struct Batches {
      address escrow;
      bytes[] callbacks;
-     uint256 additionalGas;
+     uint256[] additionalGas;
  }
```

In the `HomechainOmnichainMessenger._sendSnapshotTransfer()` function that doesn't use `callbacks`, the `additionalGas` array can be set to an array with one element.

Phuture: Fixed in PR 236

Renascence: The recommendation has been implemented. And it has been recognized that when a `TRANSFER_SNAPSHOT` message is sent, the user does not need to send additional gas.

[I-17] Performing `& type(uint32).max` before casting to `uint32` is redundant

Context:

- BaseIndex.sol#L70
- FeeMathLib.sol#L24

Description: In `BaseIndex.sol` and `FeeMathLib.sol`, `block.timestamp` is reduced to 32 bits before being cast to `uint32`:

```
## BaseIndex.sol

slot0.lastAUMAccrualTimestamp = uint32(block.timestamp & type(uint32).max);
```

```
## FeeMathLib.sol

newLastAUMAccrualTimestamp = uint32(currentTimestamp & type(uint32).max);
```

However, this is redundant as casting to `uint32` will only keep the last 32 bits of `block.timestamp`.

Recommendation:

```
## BaseIndex.sol

- slot0.lastAUMAccrualTimestamp = uint32(block.timestamp & type(uint32).max);
+ slot0.lastAUMAccrualTimestamp = uint32(block.timestamp);
```

```
## FeeMathLib.sol

- newLastAUMAccrualTimestamp = uint32(currentTimestamp & type(uint32).max);
+ newLastAUMAccrualTimestamp = uint32(currentTimestamp);
```

Phuture: Fixed in PR 223

Renascence: The recommendation has been implemented.

[I-18] Optimization when iterating over chain IDs in a `BitSet`

Context:

- ConfigBuilder.sol#L184
- ConfigBuilder.sol#L272
- ConfigBuilder.sol#L300
- HomechainOmnichainMessenger.sol#L153
- HomechainOmnichainMessenger.sol#L201

Description: Throughout the codebase, when iterating over chain IDs in a `BitSet`, `BitSet.valueAt(0, ...)` is used:

```
for (uint256 b = BitSet.find(w, 0); BitSet.hasNext(w, b);) {
    uint256 chainIndex = BitSet.valueAt(0, b);
```

This is redundant as `BitSet.valueAt(0, ...)` performs `0 | b`, which will return the same value.

Recommendation: Simply use `b` as the `chainIndex`:

```
- uint256 chainIndex = BitSet.valueAt(0, b);
+ uint256 chainIndex = b;
```

This should be applied to all the instances highlighted above.

Phuture: Fixed in PR 225

Renascence: The recommendation has been implemented.

[I-19] Preview functions in `ConfigBuilder.sol` shoud be `view` and `public`

Context:

- ConfigBuilder.sol#L202-L203
- ConfigBuilder.sol#L246-L247

Description: In `ConfigBuilder.sol`, `previewStartRebalancing()` and `previewStartReserveRebalancing()` are non-view and `external`:

```
## ConfigBuilder.sol

function previewStartRebalancing(StartRebalancingParams calldata params)
    external
```

```
## ConfigBuilder.sol

function previewStartReserveRebalancing(StartReserveRebalancingParams calldata params)
    external
```

Recommendation: Both functions should be declared `view` since they should not modify the state.

Currently, this is not possible since `IStaticPriceOracle.pricesAndBalances()`, which is called in `_currentPricesAndValuations()`, isn't declared as `view`.

Additionally, both functions should be declared `public` and used in `startRebalancing()`/`startReserveRebalancing()` directly to avoid having duplicated logic.

Phuture: Fixed in PR 225

Renascence: The recommendation has been implemented with the exception that `ConfigBuilder.previewStartRebalancing()` cannot be declared `view` since it must support non-view price sources.

[I-20] Missing `checkTotalWeight()` check in `ConfigBuilder.previewStartRebalancing()`

Context:

- ConfigBuilder.sol#L210-L211
- ConfigBuilder.sol#L154-L157

Description: `ConfigBuilder.previewStartRebalancing()` only contains the following two checks:

```
## ConfigBuilder.sol

if (keccak256(abi.encode(params.anatomy)) != anatomyHash) revert AnatomyHash();
if (!params.newAnatomy.currencyIdSets[HOME_CHAIN_INDEX].contains(RESERVE_INDEX))
revert ReserveMissed();
```

This is inconsistent with `startRebalancing()`, which performs the `RebalancingLib.checkTotalWeight()` check as well:

```
## ConfigBuilder.sol

if (keccak256(abi.encode(params.anatomy)) != anatomyHash) revert AnatomyHash();
if (!params.newAnatomy.currencyIdSets[HOME_CHAIN_INDEX].contains(RESERVE_INDEX))
revert ReserveMissed();

RebalancingLib.checkTotalWeight(params.newWeights);
```

Recommendation: Add the missing check to `previewStartRebalancing()`:

```
## ConfigBuilder.sol

  if (keccak256(abi.encode(params.anatomy)) != anatomyHash) revert AnatomyHash();
  if (!params.newAnatomy.currencyIdSets[HOME_CHAIN_INDEX].contains(RESERVE_INDEX))
revert ReserveMissed();
+ RebalancingLib.checkTotalWeight(params.newWeights);
```

Phuture: Fixed in PR 225

Renascence: The recommendation has been implemented.

[I-21] Unncessary safe cast in `RebalancingLib.previewRebalancingOrders()`

Context:

- RebalancingLib.sol#L129-L131

Description: When calculating `assets` for sell orders in `previewRebalancingOrders()`, the calculation is wrapped in a safe cast:

```
## RebalancingLib.sol

uint96 assets = valuationInfo.balances[vars.currencyIndex].mulDivDown(
    vars.currentInBase - vars.targetInBase, vars.currentInBase
).safeCastTo96();
```

However, there isn't a need to use `safeCastTo96()` here.

`valuationInfo.balances[vars.currencyIndex]` is also `uint96` and `vars.currentInBase - vars.targetInBase` will always be smaller than `vars.currentInBase`, so the result will never exceed `type(uint96).max`.

Recommendation: Perform an unchecked cast instead:

```
## RebalancingLib.sol

-  uint96 assets = valuationInfo.balances[vars.currencyIndex].mulDivDown(
+  uint96 assets = uint96(valuationInfo.balances[vars.currencyIndex].mulDivDown(
      vars.currentInBase - vars.targetInBase, vars.currentInBase
- ).safeCastTo96();
+ ));
```

Phuture: Fixed in PR 225

Renascence: The recommendation has been implemented.

[I-22] Pricing source difference between `HomechainOmnichainMessenger` and `Stargate`

Context:

- HomechainOmnichainMessenger.sol#L250

Description: To calculate the airdrop amount for Stargate transfer on the remote chain, `HomechainOmnichainMessenger` queries the `lzEndpoint` for its `sendLibrary` which is then used to fetch the relayer address and read the `dstPriceLookup`.

```
## HomechainOmnichainMessenger.sol

function _getAirdropAmount(ILzNode sendLibrary, uint16 _homeEid, SendParams calldata
sendParams, uint256 chainIndex)
    internal
    view
    returns (uint256)
{
    (uint256 cbPriceRatio,) = ILayerZeroRelayerV2Viewer(
        sendLibrary.getAppConfig(sendParams.config.eIds.at(chainIndex),
address(this)).relayer
    ).dstPriceLookup(sendParams.config.eIds.at(chainIndex));

    (uint256 sgCallbackFee,) = IStargateRouter(stargateComposer).quoteLayerZeroFee(
        _homeEid, 1, sendParams.packedRecipient, hex"", IStargateRouter.lzTxObj(0, 0,
sendParams.packedRecipient)
    );

    return sgCallbackFee.mulDivUp(CB_PRICE_RATIO_MULTIPLIER, cbPriceRatio).mulDivUp(
        sendParams.config.multipliers.at(chainIndex), MAX_BPS
    );
}
```

As Stargate is just another app built on top of LayerZero it can freely change its configuration to use another relayer. In other words, its configuration can be different than the one for `OmnichainMessenger`.

This means that its relayer can give different results for price ratios.

The finding is Informational since even if this happens:

- The price ratios should be roughly equal.
- The airdropped amount is also arbitrarily determined since a lot can change as the message is delivered to the destination chain.

Phuture: Acknowledged.

Renascence: The information has been acknowledged.

[I-23] Optimizing gas at the expense of calldata size is more expensive on L2s

Description: The protocol optimizes gas usage by storing the `keccak256` hash of the contract's state. When calling functions, the state has to be passed as arguments, which are then compared to the stored `keccak256` hash.

Such an approach makes functions cost less gas as fewer `SSTORE`s are performed. However, since the contract's state is now passed as arguments, the calldata size of transactions will be much larger.

This becomes a problem on L2s, where the cost of calldata is much more expensive than gas, as explained in:

- https://ethereum.org/en/developers/tutorials/short-abi/#cost-of-l2-transactions
- https://gist.github.com/zemse/bd65a0c853fd66216ad758cd740b8a18

As a result, transaction costs for the protocol on L2s might be much more expensive than intended.

Phuture: Acknowledged.

Renascence: The information has been acknowledged.

5 Centralization Risks

5.1 Governance

Governance should be considered fully centralized as it has full control over the protocol. By being able to upgrade protocol components, enter the rebalancing phase and setting protocol configurations, a malicious governance is able to steal all funds.

5.2 Fund Managers

Fund Managers have the privilege to call `OrderBook.executeOrder()`, which is susceptible to reeentrancy attacks. This allows a malicious fund manager to steal funds from the protocol through bypassing slippage checks while executing orders. Note that fund managers can only be set by governance, so governance must be trusted to only set non-malicious fund managers.

6 Systemic Risks

6.1 Constituent tokens

An Index consists of constituent tokens. As such, the value of the Index is derived from its underlying constituent tokens, and all risk factors that affect the constituent tokens' prices thereby affect the Index.

Moreover, the code of the constituent tokens must be trusted. For example, if a token was maliciously upgraded to always revert, there is no straightforward way for the Index to recover. Even one constituent token that misbehaves can impact the whole protocol and cause damage beyond the value that is held in that specific token.

6.2 External integrations

Phuture V2 integrates with Chainlink and Redstone for price feeds, it integrates with LayerZero for cross-chain messaging and with Stargate for cross-chain swaps.

All four of these external protocols must be fully trusted. A compromise or failure of either of them would be fatal for Phuture V2.