Ammplify: Post-Audit Technical Report

Solidity Version: `^0.8.27` / `^0.8.28`

Protocol Type: Concentrated Liquidity Market Maker (CLMM) Layer on Uniswap V3

Audited by: rubencrxz

Table of Contents

1. Protocol Overview

Ammplify is a **liquidity management protocol** that sits as a smart-contract layer on top of existing Uniswap V3 pools. It does not replace Uniswap V3 as an AMM; instead, it acts as an **intelligent aggregator and redistribution engine** that holds LP positions in Uniswap V3 on behalf of its users.

The core value proposition is threefold:

- **Aggregation.** Multiple user positions covering overlapping tick ranges are merged into a shared segment-tree representation, then deployed as a single (or small set of) Uniswap V3 position(s).
- **Reservation Market.** Privileged "Taker" participants can borrow virtual liquidity from the Maker pool and pay continuous borrow fees, creating an on-chain interest-rate market for concentrated liquidity.
- **Auto-Compounding.** Fee income from both Uniswap swap fees and Taker borrow fees can be optionally reinvested directly into the user's liquidity position.

Every pool that Ammplify manages corresponds to an existing, live Uniswap V3 pool. Ammplify holds all positions as `address(this)` in the underlying Uniswap V3 pool.

---

2. Contract Architecture

2.1 Diamond Proxy (EIP-2535)

The entire protocol is deployed as a single **EIP-2535 Diamond** contract (`SimplexDiamond`). The Diamond pattern enables a unified address with multiple upgradeable logic facets, all sharing a common storage namespace.

The Diamond is initialized in its constructor, which deploys all seven facets inline and registers them via `diamondCut`. No post-deployment initialization call is required. The Diamond supports:

Interface	Description
`IERC165`	Interface detection
`IDiamondCut`	Facet upgrade mechanism
`IDiamondLoupe`	Facet introspection
`IERC173`	Ownership

2.2 Facets

Seven facets are registered at deployment:

Facet	Role
`DiamondCutFacet`	Upgrade facets post-deployment
`DiamondLoupeFacet`	Enumerate registered functions
`AdminFacet`	Protocol configuration (fees, vaults, JIT penalties)
`MakerFacet`	Maker position lifecycle (create, adjust, remove, collect fees)

Facet	Role
`TakerFacet`	Taker position lifecycle (collateralize, open, close)
`PoolFacet`	Uniswap V3 mint callback handler
`ViewFacet`	Read-only state queries

2.3 External Contracts (Non-Diamond)

Two contracts live outside the Diamond and interact with it as external callers:

- `NFTManager` — ERC-721 wrapper that maps Ammplify `assetId`s to transferable NFTs.
- `UniV3Decomposer` — Allows migrating an existing Uniswap V3 NFT position into an Ammplify Maker position in a single transaction.

Both contracts use the **RFT (Request For Tokens)** callback pattern to handle token flows with the Diamond.

---

3. Storage Layout

3.1 Namespaced Storage Slot

Ammplify avoids storage collisions with the Diamond's own storage and any future facets by using a **deterministic ERC-7201-style storage slot**:

```
STORAGE_SLOT =
keccak256(abi.encode(uint256(keccak256("ammplify.storage.20250715")) - 1)) &
~bytes32(uint256(0xff))
             = 0x56d575db4d6456485aa5ce65d80d7d37cbb42d6dfdcfcad47c900d34e619bc00
```

All protocol state is accessed through the `Store` library, which loads this slot and returns a typed `Storage` struct.

3.2 Top-Level Storage Struct

```
struct Storage {
    AssetStore _assets;
    mapping(address poolAddr => Pool) pools;
    VaultStore _vaults;
    FeeStore _fees;
}
```

Field	Type	Description
`_assets`	`AssetStore`	Registry of all user positions (Maker and Taker)
`pools`	`mapping`	Per-pool segment tree state (nodes) and timestamp

Field	Type	Description
`_vaults`	`VaultStore`	Registry of ERC-4626 vaults used for Taker collateral
`_fees`	`FeeStore`	Protocol fee configuration and Taker collateral balances

3.3 Transient Storage

Ammplify uses **EIP-1153 transient storage** in two places:

1. **Pool Guard** (`PoolLib.POOL_GUARD_SLOT`): Stores the address of the active Uniswap V3 pool during a `mint` call, enabling the `uniswapV3MintCallback` in `PoolFacet` to verify that the call originates from the expected pool.
2. **`UniV3Decomposer.caller`**: Declared as `address private transient caller`, used as a reentrancy lock within the decomposer.

---

4. Segment Tree Data Structure

4.1 Conceptual Model

The central data structure in Ammplify is a **binary segment tree** that spans the full tick range of a Uniswap V3 pool. Each leaf in the tree corresponds to one tick spacing unit; each internal node represents an interval that is a power-of-two multiple of the tick spacing.

This design allows O(log N) range queries and updates, where N is the number of discrete tick-spacing intervals in the pool's full tick range.

4.2 Key (Node Identifier)

Each node is uniquely identified by a `Key`, a `uint48` value encoding two fields packed into 48 bits:

```
Key layout: [ width (24 bits) | base (24 bits) ]
```

- **`base`**: The leftmost tree index covered by this node (inclusive).
- **`width`**: The number of tree indices covered (always a power of two).

`Key` operations are implemented in `KeyImpl`:

- `parent()`, `children()`, `sibling()` for tree traversal.
- `isLeft()`, `isRight()`, `isLeaf()` for positional checks.
- `ticks()` to convert a Key back to Uniswap V3 `int24` tick bounds.

4.3 Tick-to-Index Mapping

Uniswap V3 uses signed 24-bit ticks (`int24`). The segment tree uses unsigned indices (`uint24`). The conversion in `TreeTickLib` is:

```
treeIndex = (tick / tickSpacing) + (rootWidth / 2)
```

The root width is the largest power of two that fits within the tick range of the pool, computed using the `msb()` (most-significant bit) function on `(-MIN_TICK / tickSpacing)`.

4.4 Node State

Each tree node (`Node`) holds two sub-structs:

`LiqNode` — Liquidity accounting:

Field	Type	Description
`mLiq`	`uint128`	Maker liquidity at this node specifically
`tLiq`	`uint128`	Taker liquidity at this node specifically
`ncLiq`	`uint128`	Non-compounding fraction of `mLiq`
`shares`	`uint128`	Total shares of compounding maker liquidity at this node
`subtreeMLiq`	`uint256`	Sum of all maker liquidity in the subtree (width-weighted)
`subtreeTLiq`	`uint256`	Sum of all taker liquidity in the subtree (width-weighted)
`subtreeBorrowedX/Y`	`uint256`	Total token amounts borrowed by takers in this subtree
`feeGrowthInside0/1X128`	`uint256`	Uniswap V3 fee growth checkpoints for this node's tick range
`borrowed`	`uint128`	Liquidity this node has borrowed from its parent
`lent`	`uint128`	Liquidity this node has lent to its children
`preBorrow` / `preLend`	`int128`	Scheduled adjustments applied during solvency resolution
`dirty`	`bool`	Flag indicating the node requires synchronization with Uniswap V3

`FeeNode` — Fee accounting:

Field	Type	Description
`takerXFeesPerLiqX128`	`uint256`	Cumulative per-unit-liquidity fees owed by takers in token0
`takerYFeesPerLiqX128`	`uint256`	Cumulative per-unit-liquidity fees owed by takers in token1
`makerXFeesPerLiqX128`	`uint256`	Cumulative per-unit-liquidity fees earned by non-compounding makers in token0
`makerYFeesPerLiqX128`	`uint256`	Cumulative per-unit-liquidity fees earned by non-compounding makers in token1
`xCFees` / `yCFees`	`uint128`	Accumulated fees earmarked for compounding (not yet converted to liquidity)

Field	Type	Description
`unclaimedMakerXFees/YFees`	`uint128`	Fees received at this node not yet distributed down the tree
`unpaidTakerXFees/YFees`	`uint128`	Fees charged to takers not yet distributed down the tree

4.5 Route and Walking

The `Route` struct defines the traversal path for a given tick range `[lowTick, highTick]`:

```
struct Route {
    uint24 rootWidth;
    Key lca;    // Lowest Common Ancestor of left and right boundary nodes
    Key left;   // Leftmost node in the canonical decomposition of the range
    Key right;  // Rightmost node in the canonical decomposition of the range
}
```

`RouteImpl.walk()` executes a two-phase traversal:

1. **Down phase**: From the root to the leaf nodes, distributing pending fees.
2. **Up phase**: From the leaf nodes back to the root, aggregating state changes.

Each node is categorized as either **visited** (part of the target range) or **propagated** (an ancestor that must be kept consistent). The maximum tree depth is bounded by `MAX_WIDTH_LENGTH = 21`, corresponding to Uniswap V3's maximum tick range.

---

5. Position and Asset System

5.1 Asset

A user's position is represented as an `Asset` stored in the `AssetStore`:

```
struct Asset {
    address owner;
    address poolAddr;
    int24 lowTick;
    int24 highTick;
    LiqType liqType;
    uint8 xVaultIndex;
    uint8 yVaultIndex;
    uint128 liq;        // Original (target) liquidity
    uint128 timestamp;  // Last modification timestamp (for JIT penalty)
    mapping(Key => AssetNode) nodes;
}
```

Asset IDs are assigned sequentially from a global counter (`nextAssetId`), starting at 1. Each address is limited to **16 concurrent assets** (`MAX_ASSETS_PER_OWNER`).

5.2 AssetNode

For every tree node that covers a portion of the asset's tick range, an `AssetNode` is stored:

```
struct AssetNode {
    uint128 sliq;           // Shares (MAKER) or direct liquidity (MAKER_NC/TAKER)
    uint256 fee0CheckX128;  // Per-liq fee checkpoint at last interaction
    uint256 fee1CheckX128;
}
```

The checkpoint pattern mirrors Uniswap V3's fee accounting: fees accrued to an asset are `(currentRate - checkpoint) * sliq`.

---

6. Liquidity Types

Ammplify defines three distinct liquidity types via the `LiqType` enum:

6.1 `MAKER` — Compounding Maker

- User provides liquidity to the pool.
- Earned fees (both Uniswap swap fees and Taker borrow fees) are **automatically reinvested** as additional liquidity.
- Ownership is tracked via a **share system**: upon depositing, the user receives `shares` proportional to the pool's total value at the time of deposit, including any uncompounded fees.
- When withdrawing, the user receives a share-proportional amount of both liquidity and the pending fee balances (`xCFees`, `yCFees`).

6.2 `MAKER_NC` — Non-Compounding Maker

- User provides liquidity to the pool.
- Earned fees are **tracked separately** via per-liq-rate checkpoints and can be claimed by calling `collectFees()`.
- Ownership is tracked with a direct 1:1 liquidity amount (`sliq == liq`), no share arithmetic.

6.3 `TAKER` — Taker (Borrower)

- A **permissioned** position type: only addresses with `AmmplifyAdminRights.TAKER` can open Taker positions.
- The Taker effectively borrows liquidity from the Maker pool, withdrawing tokens from Uniswap V3 and becoming obligated to return them.
- The Taker must deposit **collateral** (tracked in `FeeStore.collateral`) and provide a **freeze price** (`freezeSqrtPriceX96`) that determines the initial collateral requirement, stored in ERC-4626 vaults.
- Takers are charged continuous borrow fees based on the utilization rate of the pool, computed by the `SmoothRateCurveLib`.
- Upon position removal, the collateral stored in the vault is released and net balances are settled.

6.4 Minimum Liquidity Thresholds

Type	Minimum Liquidity
`MAKER` / `MAKER_NC`	`1e6`
`TAKER`	`1e12` (1000× higher)

---

7. Walker System

The walker system is the protocol's core execution engine. Every state-modifying operation on a position triggers a tree traversal that applies fees, modifies liquidity, and marks nodes as dirty for synchronization with Uniswap V3.

7.1 `WalkerLib` — Orchestrator

`WalkerLib.modify()` is the single entry point for all state-modifying walks. It:

1. Converts `int24` ticks to tree indices.
2. Constructs a `Route`.
3. Calls `route.walk()` with two callbacks: `down` and `up`.

Ordering invariant: **`FeeWalker.down` → `FeeWalker.up` → `LiqWalker.up`**. Fee state must be settled before liquidity changes are applied.

7.2 `FeeWalker` — Fee Distribution and Rate Calculation

Down phase:

- Distributes `unpaidTakerXFees` and `unclaimedMakerXFees` from each visited/propagated node into its children (`childSplit`).
- At leaf nodes, distributes into the per-liq-rate accumulators (`takerXFeesPerLiqX128`, `makerXFeesPerLiqX128`).
- Accumulates maker liquidity prefixes (`mLiqPrefix`, `tLiqPrefix`) for visited-but-not-target nodes.

Up phase (visited nodes):

- Calls `chargeTrueFeeRate()`: computes the actual borrow fee rate for this column using the `SmoothRateCurveConfig` and the column's utilization ratio (`totalTLiq / totalMLiq`), multiplied by the elapsed time since the pool's last update. The total column borrowed amounts are computed via `computeBorrows()` (which uses the **geometric mean tick** of the range).
- Stores the resulting per-liq rates in the node's `FeeNode` and distributes a portion to children via another `childSplit`.

Up phase (propagated nodes):

- Infers the fee rate by **simple averaging** of left and right child column rates.
- Applies the inferred rate to its own liquidity.
- Resets the propagated col rates and removes itself from the prefix.

Phase transitions (`FeeWalker.phase`): During `LEFT_UP → RIGHT_UP`, the rates computed for the left column are saved to `lcaLeftCol*` fields and restored at the start of `RIGHT_UP` to enable correct root propagation.

7.3 `LiqWalker` — Liquidity Modification and Solvency

Up phase:

1. **`compound()`**: Collects accrued Uniswap V3 swap fees for this node's tick range, splits them between compounding and non-compounding makers, and converts compounding fees into additional liquidity if the amount exceeds the `compoundThreshold`.
2. **`modify()`**: Applies the user's target liquidity change to the node's `LiqNode` and `AssetNode`. Behavior differs by `LiqType`:
   - `MAKER`: Share-based arithmetic.
   - `MAKER_NC`: Direct delta arithmetic.
   - `TAKER`: Borrow tracking using geometric mean for fee amounts and current price for token balances.
3. **`solveLiq()`**: Verifies `node.liq.net() >= 0` (where `net = borrowed + mLiq - tLiq - lent`). If negative, schedules a borrow from the parent node via `preBorrow` and `preLend` fields on the parent and sibling. Root nodes must always be solvent.

7.4 `PoolWalker` — Uniswap V3 Synchronization

After `WalkerLib.modify()` completes, `PoolWalker.settle()` performs a second full-tree walk. For every node marked `dirty`:

- Reads the **actual current liquidity** of that position from Uniswap V3 (`PoolLib.getLiq`).
- Computes the target liquidity as `node.liq.net()`.
- Issues `mint()` or `burn()` to Uniswap V3 accordingly.
- Clears the `dirty` flag.

7.5 `Data` — Shared Walk Context

A `Data` struct is created once per operation and threaded through all walker callbacks:

```
struct Data {
    address poolAddr;
    bytes32 poolStore;    // Assembly slot for Pool storage
    bytes32 assetStore;   // Assembly slot for Asset storage
    uint160 sqrtPriceX96; // Current Uniswap V3 price at time of Data creation
    uint128 timestamp;    // Pool timestamp before update (used for time-weighted
fees)
    LiqData liq;
    FeeData fees;
    int256 xBalance;      // Accumulated token0 net flow (positive = user pays)
    int256 yBalance;      // Accumulated token1 net flow
}
```

`DataImpl.make()` performs a **price slippage check** at creation time: `currentSqrtPriceX96` must fall within `[minSqrtPriceX96, maxSqrtPriceX96]`. It also updates `pool.timestamp` to `block.timestamp`, recording the time of the operation for fee accrual calculations.

`computeBorrows` uses the geometric mean tick of the range (midpoint of `[lowTick, highTick]`) to compute stable, price-independent token amounts for fee accounting.

`computeBalances` uses the current `sqrtPriceX96` to compute actual token transfer amounts.

---

8. Fee System

8.1 SmoothRateCurve

Both the borrow fee rate and the fee split weights are computed using `SmoothRateCurveLib`, parameterized by `SmoothRateCurveConfig`:

```
struct SmoothRateCurveConfig {
    uint128 invAlphaX128;  // Controls curve steepness
    uint64  betaX64;       // Minimum rate at 0% utilization
    uint64  maxUtilX64;    // Utilization cap (beyond which rate is maximum)
    uint64  maxRateX64;    // Maximum rate
}
```

Default borrow rate parameters at initialization:

- **5% annual rate at 0% utilization**
- **~30% at 60% utilization**
- **95% at 120% utilization cap** (`maxUtilX64 = 120%`)

8.2 Fee Channels

There are two distinct fee channels for Maker positions:

Channel	Source	Applicable to
Uniswap swap fees	Collected via `PoolLib.collect()` during the `compound()` step	Both MAKER and MAKER_NC
Taker borrow fees	Charged continuously per unit time × utilization rate	Both MAKER and MAKER_NC

For MAKER positions, both channels accumulate into `xCFees`/`yCFees` and are converted to liquidity upon the next interaction. For MAKER_NC positions, borrow fees accrue into `makerXFeesPerLiqX128` and are claimed via checkpoint arithmetic.

8.3 Fee Split

When fees are propagated from a parent node to its left and right children, the split ratio is determined by `getLeftRightWeights()`. This function computes a utilization-weighted score for each child using the `splitConfig SmoothRateCurveConfig`, with higher-utilization subtrees receiving a larger share of the fee income.

8.4 JIT Penalty

To disincentivize just-in-time (JIT) liquidity provision, positions that are removed within `jitLifetime` seconds of their last modification timestamp pay a `jitPenaltyX64` fraction penalty on the returned tokens. Applied in `FeeLib.applyJITPenalties()`, which is called within `removeMaker()` and `adjustMaker()` after settlement.

The penalty is applied as a **multiplicative reduction** on the returned token amounts:

```
xBalanceOut = xBalance * jitPenaltyX64 / 2^64
```

Default values for `jitLifetime` and `jitPenaltyX64` are not initialized in `FeeLib.init()` (both start at 0), meaning JIT protection is disabled by default and must be explicitly configured by the owner.

8.5 Compound Threshold

To avoid gas-inefficient micro-compounding, a `compoundThreshold` (in equivalent liquidity units) controls whether accumulated `xCFees`/`yCFees` are worth converting into liquidity. Default: `1e12`.

---

9. Vault System

9.1 Purpose

Vaults hold Taker collateral. When a Taker opens a position, a collateral amount (determined by `freezeSqrtPriceX96`) is computed and deposited into the vault. This collateral is returned when the position is closed.

9.2 ERC-4626 Integration (`VaultE4626`)

The only currently supported vault type is `E4626` (ERC-4626 yield-bearing vaults). Each vault wrapper:

- Tracks total vault shares (`totalVaultShares`) in the underlying ERC-4626.
- Maintains per-position share accounting (`shares[assetId]`) to attribute ownership fractions.
- **Batches operations**: deposit and withdraw amounts are accumulated via `VaultTemp` and committed in a single ERC-4626 interaction during `commit()`. Net-outs (simultaneous deposit and withdraw) save vault entry/exit fees.
- A `highWaterMark` field is stored but not referenced in the current implementation.

9.3 Vault Lifecycle

Each token can have two indexed vaults: an **active** vault and a **backup** vault. The `VaultProxy` struct routes all operations to the active vault, optionally mirroring to the backup during migration.

Admin functions:

- `addVault`: Registers a vault as primary then backup.
- `removeVault`: Removes the backup vault (only if balance < `BALANCE_DE_MINIMUS = 10`).
- `swapVault`: Atomically promotes the backup to active and demotes the active to backup without moving funds.
- `transferVaultBalance`: Moves an explicit amount from one vault to another.

---

10. Integration Layer

10.1 NFTManager

`NFTManager` is a standalone ERC-721 contract that wraps Ammplify positions as transferable NFTs.

Ownership model: When a user mints via `NFTManager`, the `NFTManager` contract becomes the `asset.owner` in the Diamond, while the user holds the ERC-721 token. Transfer of the NFT does not automatically update `asset.owner` in the Diamond; the NFT conveys the right to call operations through the NFTManager.

RFT callback: `NFTManager` implements `IRFTPayer.tokenRequestCB`. During `mintNewMaker`, the current token requester (`_currentTokenRequester`) is set to `msg.sender` so the callback correctly identifies who to pull tokens from.

Token URI: Fully on-chain SVG metadata encoded in Base64.

Key functions:

- `mintNewMaker`: Creates a Maker position and mints an NFT in one transaction.
- `decomposeAndMint`: Accepts a Uniswap V3 NFT, decomposes it, and mints an Ammplify NFT.
- `burnAsset`: Collects fees, removes the Maker position, and burns the NFT.
- `collectFees`: Claims fees without closing the position.

10.2 UniV3Decomposer

`UniV3Decomposer` converts a Uniswap V3 NFT position into an Ammplify Maker position:

1. Reads position metadata from `INonfungiblePositionManager`.
2. Transfers the Uniswap V3 NFT to itself.
3. Calls `decreaseLiquidity` with the full liquidity amount (no slippage protection: `amount0Min = 0`, `amount1Min = 0`).
4. Calls `collect` to withdraw all tokens and fees.
5. Burns the original Uniswap V3 NFT.
6. Looks up the corresponding pool via `IUniswapV3Factory`.
7. Calls `MAKER.newMaker()` with `liquidity - liquidityOffset`, where `liquidityOffset` is computed to account for rounding in the conversion.

`calculateLiquidityOffset`: Computes a dynamic offset based on tick range width:

```
offset = (2^96 * 42) / (sqrtPrice(high) - sqrtPrice(low))
```

This formula yields a larger offset for narrow ranges (where small liquidity differences matter more) and a smaller offset for wide ranges. The constant `42` is the `LIQUIDITY_OFFSET` used as numerator base.

Reentrancy guard: Implemented via `address private transient caller`. The lock is set to `msg.sender` at the start and reset at the end of `decompose()`.

---

11. Access Control

11.1 Ownership (`AdminLib` / `TimedAdminFacet`)

The Diamond uses a two-step **timed ownership transfer** pattern from `Commons/Util/TimedAdmin.sol`:

- `transferOwnership`: Proposes a new owner; effective after `getDelay(true) = 3 days`.
- `acceptOwnership`: The proposed owner accepts.

11.2 Rights System

Beyond the owner, additional permissioned roles are managed via a timed rights system:

- `submitRights(address, uint256)`: Proposes granting a bitmask of rights to an address; effective after `getDelay(true) = 3 days`.
- `acceptRights`: The proposed grantee accepts.
- `removeRights`: Scheduled revocation after `getDelay(false) = 1 day`.
- `vetoRights`: Immediate revocation by the owner.

Currently, only one rights flag is defined:

```
uint256 public constant TAKER = 0x1;
```

Only addresses with `TAKER` rights can call `newTaker()` and `withdrawCollateral()`.

---

12. Mathematical Primitives and Precision

12.1 Fixed-Point Notation

All fee rates and liquidity calculations use explicit fixed-point notation with suffix indicating the denominator:

- `X128` → value × 2^128 (Q128 format, used for per-liq fee rates)
- `X96` → value × 2^96 (Uniswap V3's native `sqrtPriceX96` format)
- `X64` → value × 2^64 (used for utilization and rate curve outputs)

12.2 `FullMath`

A local copy of the Uniswap V3 `FullMath` library providing overflow-safe `mulDiv` operations for 256-bit intermediate values. Extended with:

- `mulX128(a, b, roundUp)` — multiply `a` by a Q128 fixed-point value `b`.
- `mulX64(a, b, roundUp)` — multiply `a` by a Q64 fixed-point value `b`.
- `mulX256(a, b, roundUp)` — multiply `a` by a Q256 fixed-point value `b`.
- `mulDivX256` — `a * b / c` returning a Q256 result.

12.3 Rounding Convention

- Fee rates charged **to takers** round **up** (conservative: ensures the protocol is never underpaid).
- Fee rates earned **by makers** round **down** (conservative: ensures makers cannot be overpaid).
- Liquidity amounts round **down** (prevents minting more liquidity than tokens support).

12.4 Overflow Protection in Fee Accumulation

`FeeWalker.add128Fees()` handles the case where accumulated fees would overflow `uint128`. When `a + b > type(uint128).max`, the excess is immediately credited to the current operation's `data.xBalance` / `data.yBalance`, effectively paying it out to the user in the current transaction rather than losing it.

12.5 Assembly Usage

Assembly is used in three specific contexts:

1. **`Store.load()`**: Loads the namespaced storage struct by its deterministic slot.
2. **`DataImpl.node()` / `DataImpl.assetNode()`**: Reconstructs storage pointers from `bytes32` slot addresses stored in the `Data` struct.
3. **`VaultLib.getVault()`**: Extracts an ERC-4626 vault's storage pointer from the `VaultStore`.
4. **`WalkerLib.toRaw()` / `toData()`**: Unsafe pointer casts between `Data memory` and `bytes memory` to avoid copies when passing through function pointer callbacks (valid because the ABI layout is identical).

---

13. External Dependencies

Dependency	Usage
`v3-core` (Uniswap V3)	Pool interface (`IUniswapV3Pool`), tick math, sqrt price math
`v4-core` (Uniswap V4)	Imports `TickMath` and `SqrtPriceMath` from v4-core library
`openzeppelin-contracts`	`ReentrancyGuardTransient`, `TransientSlot`, `SafeERC20`, `IERC4626`, `ERC721`, `Ownable`, `SafeCast`
`Commons` (internal)	Diamond infrastructure, `AdminLib`, `TimedAdminFacet`, `RFTLib`, `SmoothRateCurveLib`, `FullMath`, `TransferHelper`

Note on version mixing: `TickMath` is imported from `v3-core` in `UniV3Decomposer` and from `v4-core` in `Pool.sol`. Both provide the same `getSqrtPriceAtTick` semantics in the versions used, but represent different library sources.

---

14. Data Flow: End-to-End Operation

The following describes the complete call path for `MakerFacet.newMaker()`:

```
1. MakerFacet.newMaker(recipient, poolAddr, lowTick, highTick, liq, isCompounding,
...)
   │
   ├── PoolLib.getPoolInfo(poolAddr)           // Read token addresses, tick
spacing, tree width from Uniswap V3
   ├── AssetLib.newMaker(...)                  // Allocate new assetId, write
Asset to storage
   ├── DataImpl.make(pInfo, asset, ...)        // Snapshot current sqrt price,
```

```
update pool.timestamp, build Data
   │     └── price slippage check
   │
   ├── WalkerLib.modify(pInfo, lowTick, highTick, data)
   │     └── route.walk(down, up, phase, data)
   │           ├── [DOWN] FeeWalker.down() on each node    // Distribute pending
fees down the path
   │           └── [UP]   FeeWalker.up()  on each node    // Calculate & charge
borrow fee rates
   │                      LiqWalker.up()  on each node    // Compound, modify
liquidity, solve solvency
   │                        ├── compound()                // Collect Uniswap fees,
reinvest if above threshold
   │                        ├── modify()                  // Apply share or direct
liq delta; update data.xBalance/yBalance
   │                        └── solveLiq()               // Borrow from parent if
netLiq < 0
   │
   ├── RFTLib.settle(msg.sender, tokens, balances, rftData)  // Pull tokens from
user (xBalance/yBalance > 0)
   │
   └── PoolWalker.settle(pInfo, lowTick, highTick, data)
         └── route.walk(down=noop, up=updateLiq, phase=noop, data)
               └── for each dirty node:
                     ├── PoolLib.getLiq()           // Read actual Uniswap V3
position liquidity
                     ├── PoolLib.mint() or burn()   // Sync Uniswap V3 to target
                     └── dirty = false
```

Token balances flow as follows:

- Positive `data.xBalance` / `data.yBalance` at the end of the walker phase → tokens are pulled **from** the user.
- Negative values → tokens are sent **to** the user.
- `RFTLib.settle()` performs the net transfer using either direct ERC-20 transfers or a payer callback pattern.

---

15. My Findings

15.1 View.sol::queryAssetBalances Provides Inaccurate Information Potentially Leading to State-Changing Actions Under False Assumptions

Summary

The `ViewWalker.down()` function contains a variable assignment error when naming unpaid fees. This corrupts fee calculations returned by `queryAssetBalances()`. Since this view function is intended by the protocol to be trustworthy as a way to check a user's position balances by providing accurate information, returning wrong values can directly mislead users' decisions leading to state-changing function calls based on inaccurate information.

Vulnerability Details

The bug originates in the fee distribution logic inside View.sol walker's contract (View.sol#L259-L262):

```
function down(Key key, bool visit, ViewData memory data) internal view {
    // ...
    data.leftChildUnpaidX = leftPaid;
    data.rightChildUnpaidY = unpaidX - leftPaid;  // ← Bug: should be
rightChildUnpaidX
    data.leftChildUnclaimedX = leftEarned;
    data.rightChildUnclaimedX = unclaimedX - leftEarned;
    // ...
}
```

The assignment should target `rightChildUnpaidX`, not `rightChildUnpaidY`. This bug injects incorrect values into the walk down.

That corrupted data is then returned through the full `queryAssetBalances` chain (View.sol#L80-L101):

```
function queryAssetBalances(uint256 assetId)
    external view
    returns (int256 netBalance0, int256 netBalance1, uint256 fees0, uint256 fees1)
{
    Asset storage asset = AssetLib.getAsset(assetId);
    PoolInfo memory pInfo = PoolLib.getPoolInfo(asset.poolAddr);
    ViewData memory data = ViewDataImpl.make(pInfo, asset);
    ViewWalkerLib.viewAsset(pInfo, asset.lowTick, asset.highTick, data);
    if (asset.liqType == LiqType.TAKER) {
        netBalance0 = int256(vaultX) - int256(data.liqBalanceX);
        netBalance1 = int256(vaultY) - int256(data.liqBalanceY);
        fees0 = data.earningsX;  // ← corrupted
        fees1 = data.earningsY;  // ← corrupted
    }
}
```

Inside `ViewWalkerLib` (Lib.sol#L54-L63), the call path is: `queryAssetBalances()` → `ViewWalkerLib.viewAsset()` → `ViewRouteImpl.walkDown()` → `ViewWalkerLib.down()` → `ViewWalker.down()`.

Impact

The function `queryAssetBalances()` is the protocol's main entry point for users to understand their current liquidity and fee positions. By design, this function must be trustworthy. Returning corrupted balances due to the bug can lead to:

- **Misleading fee accrual:** Users may see inflated fees in one token, leading them to close positions prematurely to realize non-existent rewards.

- **Collateral mismanagement:** A taker may withdraw collateral believing a positive net balance while the position is undercollateralized.
- **Closing positions based on inaccurate data:** Both Makers and Takers rely on `queryAssetBalances()` for critical financial decisions. Incorrect output may lead users to execute state-changing functions under false assumptions, exposing them and the protocol to economic inconsistencies.

Recommended Mitigation

Correct the assignment in `View.sol::down()`:

```
data.leftChildUnpaidX = leftPaid;
- data.rightChildUnpaidY = unpaidX - leftPaid;
+ data.rightChildUnpaidX = unpaidX - leftPaid;
data.leftChildUnclaimedX = leftEarned;
data.rightChildUnclaimedX = unclaimedX - leftEarned;
```

15.2 Admin.sol::transferVaultBalance Function is Hardcoded to Non-Existent User ID, Breaking Vault Migration

Summary

The `transferVaultBalance` function in Admin.sol uses a hardcoded `TAKER_VAULT_ID` (80085) instead of accepting a user `assetId`, making vault migration impossible for real taker positions and breaking critical admin functionality.

Vulnerability Details

In `Vault.sol::transfer()`, the third parameter `userId` determines whose funds get moved (Vault.sol#L106-L114):

```
function transfer(address fromVault, address toVault, uint256 userId, uint256
amount) internal {
    VaultPointer memory from = getVault(fromVault);
    from.withdraw(userId, amount);
    from.commit();
    VaultPointer memory to = getVault(toVault);
    to.deposit(userId, amount);
    to.commit();
    emit VaultTransfer(fromVault, toVault);
}
```

In `Admin.sol::transferVaultBalance`, this `userId` parameter is hardcoded to `TAKER_VAULT_ID` (Admin.sol#L131-L134):

```
function transferVaultBalance(address fromVault, address toVault, uint256 amount)
external {
```

```
        AdminLib.validateOwner();
        VaultLib.transfer(fromVault, toVault, TAKER_VAULT_ID, amount);
    }
```

Broken flow: A taker gets a unique `assetId` after creating a position. Funds are stored per user via `Vault.sol::deposit` using their actual `assetId`. When admin needs to migrate taker funds to another vault via `transferVaultBalance`, the function transfers funds for `TAKER_VAULT_ID`, which is never used as a real user ID in the protocol (it is a hardcoded constant). Real user funds from takers remain stuck in the source vault with no way to migrate them.

Impact

Complete loss of critical admin functionality: the function transfers from a non-existent user balance, making vault migration impossible and effectively locking taker balances.

Recommended Mitigation

Add `userId` as an input parameter to specify which user's funds the admin intends to transfer:

```
function transferVaultBalance(
    address fromVault,
    address toVault,
+   uint256 userId,
    uint256 amount
) external {
    AdminLib.validateOwner();
    VaultLib.transfer(
        fromVault,
        toVault,
-       TAKER_VAULT_ID,
+       userId,
        amount
    );
}
```
