Ammplify

Ammplify is a protocol designed to enhance the yield and capital efficiency of liquidity providers in concentrated AMMs such as Uniswap v3. Its main goal is to amplify LP returns and quantify risk through intelligent fee reinvestment and liquidity pooling, without altering the user's original range or AMM setup.

Core Concepts

1. Bayesian Auto-Compounding

Traditional auto-compounders reinvest fees uniformly across the entire LP range. Ammplify improves on this with a Bayesian approach — it compounds only into the ticks that have recently generated fees, based on the assumption that past profitable ticks are statistically more likely to continue earning. This makes the process exponentially cheaper and more efficient, while the user's original LP remains untouched (only the compounded portion is exposed to impermanent loss).

2. Pooled Liquidity Lending

Each LP position in Uniswap v3 is unique, which makes lending and borrowing difficult. Ammplify solves this by splitting positions into fungible pieces of liquidity, pooling them into shared "buckets." This allows users to lend or borrow portions of liquidity while maintaining their own custom ranges. The protocol distinguishes between:

Maker liquidity → regular LP deposits that can be lent out.

Taker liquidity → borrowed liquidity that pays usage fees to makers.

Together, these features let liquidity providers earn both AMM swap fees and additional lending yield, while keeping their capital flexible.

Technical Details

- Smart Contract Architecture

Ammplify is implemented following the Diamond Standard (EIP-2535) and is composed of several facets that split functionality modularly:

Facet Function MakerFacet Opens/closes LP positions TakerFacet Manages borrowing and collateral PoolFacet Interfaces with underlying AMM pools ViewFacet Exposes on-chain state and analytics AdminFacet Handles governance and role management

- WalkerLib and the Segment Tree

The protocol's core logic lives in WalkerLib, which manages fee distribution and liquidity accounting through a segment tree structure. Each node in the tree represents a range of ticks, containing both maker and taker liquidity data. When liquidity is modified, the protocol "walks" down and up the tree:

Walk down → distributes unpaid/ unclaimed fees across subtrees.

Walk up → recalculates rates, compounds fees, and solves borrow balances recursively.

This tree design allows updates to propagate efficiently across overlapping LPs — a single compounding action can benefit multiple users whose positions share the same range.

- Assets and AssetNodes

User positions are stored as Assets, each tracking their balances, node participation, and fees. Each node used by a user is represented as an AssetNode, which records the share of liquidity and accumulated rewards for that tick range. When the tree is updated through WalkerLib, these AssetNodes are synchronized accordingly.

My Findings

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: Link: https://github.com/sherlock-audit/2025-09-ammplify/blob/main/Ammplify/src/walkers/View.sol#L259-L262

```
function down(Key key, bool visit, ViewData memory data) internal view {
 .
 .
```

```
 .
    data.leftChildUnpaidX    = leftPaid;
@> data.rightChildUnpaidY   = unpaidX - leftPaid;
    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: Link: https://github.com/sherlock-audit/2025-09-ammplify/blob/main/Ammplify/src/facets/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
    }
}
```

And inside ViewWalkerLib: Link: https://github.com/sherlock-audit/2025-09-ammplify/blob/main/Ammplify/src/walkers/Lib.sol#L54-L63

```
function viewAsset(PoolInfo memory pInfo, int24 lowTick, int24 highTick, ViewData
memory data) internal view {
    uint24 low = pInfo.treeTick(lowTick);
    uint24 high = pInfo.treeTick(highTick) - 1;
    Route memory route = RouteImpl.make(pInfo.treeWidth, low, high);
@>    ViewRouteImpl.walkDown(route, down, phase, toRaw(data));
}

function down(Key key, bool visit, bytes memory raw) internal view {
@>  ViewWalker.down(key, visit, toData(raw));
}
```

Complete call path: 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, as its output provides critical information. Returning corrupted balances due to the bug can lead to serious effects, such as:

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 that it has a positive net balance, but in reality the position is undercollateralized. Closing positions based on inaccurate data. Both Makers and Takers rely on queryAssetBalances() for critical financial decisions. An incorrect output in this function may lead users to execute state-changing functions under false assumptions, exposing them and the protocol to economic inconsistencies.

Tools Used

Manual review.

Recommended Mitigation

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

```
function down(Key key, bool visit, ViewData memory data) internal view {
        // On the way down, we accumulate the prefixes and claim fees.
        Node storage node = data.node(key);
        AssetNode storage aNode = data.assetNode(key);
.
.
.
data.leftChildUnpaidX    = leftPaid;
- data.rightChildUnpaidY = unpaidX - leftPaid;
+ data.rightChildUnpaidX = unpaidX - leftPaid;
data.leftChildUnclaimedX = leftEarned;
data.rightChildUnclaimedX = unclaimedX - leftEarned;
.
.
.
```

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 is critical as it determines whose funds get moved: Link: https://github.com/sherlock-audit/2025-09-ammplify/blob/main/Ammplify/src/vaults/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 : Link: https://github.com/sherlock-audit/2025-09-ammplify/blob/main/Ammplify/src/facets/Admin.sol#L131-L134

```
function transferVaultBalance(address fromVault, address toVault, uint256 amount)
external {
          AdminLib.validateOwner();
@>    VaultLib.transfer(fromVault, toVault, TAKER_VAULT_ID, amount);
```

Flow that will be broken:

Taker gets a unique assetId after creating a position. Funds are stored per user via Vault.sol::deposit using their actual assetId. There is a need to migrate taker funds to another vault, so admin role calls transferVaultBalance. The function tries to transfer 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 the taker remain stuck in the source vault with no way to migrate them.

Impact

This vulnerability leads to a complete loss of critical admin functionality, having a function that transfers from non-existent user balance, making vault migration impossible and locking taker balances.

Tool used

Manual Review.

Recommended Mitigations

Consider adding userId as an input parameter to specify which user's funds admin has to transfer:

```
function transferVaultBalance
      (address fromVault,
```

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