certora

Security Assessment
Final Report

Likwid Protocol

May 2025

Prepared for Likwid.fi

certora

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Initial Commit Hash	Platform
Likwid.fi	Github repository	b60b094	EVM

Project Overview

This document describes the specification and verification of **Likwid.fi** using manual code review. The work was undertaken from **April 25th, 2025** to **May 14th, 2025**.

The team performed a manual audit of all the Solidity contracts. During the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

Likwid.fi is a decentralized derivatives platform built on Uniswap V4. It enables users to long, short, and trade ERC-20 token pairs without oracles or direct counterparties, using pooled liquidity and collateral-backed positions. Lenders can provide liquidity to the protocol to earn yield from traders and borrowers. Unlike Uniswap V4's concentrated liquidity model, Likwid.fi uses a traditional constant product AMM (x*y=k) to price assets. The protocol leverages Uniswap V4's infrastructure—such as hooks and singleton pools—to implement custom trading logic and dynamic fees while maintaining full decentralization and capital efficiency.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	1	1	1
High	9	9	9
Medium	11	11	5
Low	4	4	2
Informational	2	2	1
Total	27	27	18

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
C-01	DoS due to underflow in `getInterestReserves()` by malicious liquidity minting to `PairPoolManager`	Critical	Fixed
H-01	Pool Truncated Reserve are updated only if the Interest is opened	High	Fixed
H-02	`swapMirror()` allows frontrunning to steal user funds by specifying arbitrary sender and recipient	High	Fixed
H-03	Missing slippage protection in `swapMirror()`	High	Fixed
H-04	`removeLiquidity()` is susceptible to sandwich attacks due to missing slippage protection	High	Fixed
H-05	Liquidity removals and transfers can be griefed	High	Fixed
H-06	`MarginRouter::exactOutput` is vulnerable to sandwich attacks	High	Fixed
H-07	`MarginPositionManager::modify` uses wrong amount when decreasing `position.marginAmount`	High	Fixed

H-08	Miscalculated margin limits allow unsafe borrowing	High	Fixed
H-09	Decreasing margin position using modify may withdraw less than intended funds	High	Fixed
M-01	Donating lending pool tokens deflates Interest accrual for other users	Medium	Fixed
M-02	Position transfer can be frontrun and blocked via Denial-of-Service	Medium	Acknowledged
M-03	Broken ERC6909 compliance due to events emitting incorrect (unscaled) amounts	Medium	Fixed
M-04	`MarginPositionManager::close` function doesn't check the position is healthy	Medium	Fixed
M-05	Liquidations prioritize caller and protocol profit over pool solvency	Medium	Fixed
M-06	User is not enforced to provide `msg.value` on native swaps in `MarginRouter`	Medium	Fixed
M-07	Retroactive application of updated protocol Interest Fee leads to unfair distribution	Medium	Acknowledged
M-08	Unannounced change in `liquidationMarginLevel` can trigger unfair liquidations	Medium	Acknowledged

M-09	`_getPriceDegree` may wrongly return high fee for small swap	Medium	Acknowledged
M-10	Anyone can open position for another user, which increases victim's risk	Medium	Acknowledged
M-11	Improper Interest accrual on re-enabling Interest for pool	Medium	Partially fixed
L-01	`mirrorTokenManager` can add new managers but can't revoke old ones	Low	Acknowledged
L-02	`estimatePNL` always returns wrong value for borrow positions	Low	Fixed
L-03	`interestOperator` mapping becomes outdated if `statusManager` is changed in the pool manager	Low	Acknowledged
L-04	Hardcoded global slippage tolerance (`maxSliding`) limits user control and may cause liquidity provisioning failures	Low	Fixed

Critical Severity Issues

C-01 DoS due to underflow in `getInterestReserves()` by malicious liquidity minting to `PairPoolManager`

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: MarginLiquidity.sol#L301-L302 MarginLiquidity.sol#L371-L372	Status: Fixed	

Description: An attacker can permanently break a newly created pool by back-running its creation and minting a small amount of liquidity to the `PairPoolManager` address. This causes an accounting inconsistency that leads to an underflow in `getInterestReserves()`, which is a core function used by `_update()`.

The issue arises because the protocol tracks the retained LP supply as `balanceOf(pairPoolManager, levelId)`. However, there is nothing preventing malicious users from minting tokens directly to the `PairPoolManager` for a `levelId`, thus inflating `retainSupply0` or `retainSupply1` beyond `totalSupply` due to the double accounting of retain supplies:

```
JavaScript
_mint(caller, poolManager, uPoolId, amount);
_mint(caller, poolManager, levelId, amount);
_mint(caller, receiver, levelId, amount); //@audit Double accounting of retain supply
```

This leads to an underflow in the expression `totalSupply - retainSupplyX`, which causes a revert, DoS-ing the pool:

```
Unset
(uint256 totalSupply, uint256 retainSupply0, uint256 retainSupply1) =
_getPoolSupplies(pairPoolManager, uPoolId);
if (totalSupply > 0) {
       reserve0 = Math.mulDiv(totalSupply - retainSupply0, status.reserve0(), totalSupply);
       reserve1 = Math.mulDiv(totalSupply - retainSupply1, status.reserve1(), totalSupply);
}
```

Exploit Scenario:

1. Alice creates a new pool using `PoolManager.initialize()`.
2. Bob observes this transaction in the mempool and backruns the pool initialization by calling `PairPoolManager.addLiquidity()` to mint `1 wei` of liquidity to the `PairPoolManager` for the new pool's ID.

From this point on, any operation that relies on `getInterestReserves()` will revert due to underflow, effectively bricking the pool permanently.

The attack is extremely cheap and easy to automate - backrun newly created pools that use the Likwid.Fi hook. The pool creators or users have no recovery path.

Recommendations: Prevent users from minting liquidity tokens to the `PairPoolManager` address by enforcing stricter checks in `addLiquidity()` and `_mint()` to ensure LP tokens cannot be sent to system-critical addresses.

Customer's response: Fixed in db6c5.

Fix Review: Fix confirmed.

High Severity Issues

H-01 Pool Truncated Reserve are updated only if the Interest is opened

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: PoolStatusManager.sol #L459-L460	Status: Fixed	

Description: The `_transform` function—which updates `truncatedReserves` and `status.blockTimestampLast`—is only invoked via `_updateInterests`. However, `_updateInterests` is only called within `PoolStatusManager::setBalances` **if interest is enabled**.

This means that when **interest is disabled**, the protocol **skips critical updates** to internal reserve tracking and timestamps. As a result, an exploiter could manipulate pool reserves without those manipulations being reflected in the system's internal accounting.

Recommendations: Ensure `_transform` is called unconditionally within `setBalances`, regardless of whether interest is enabled. This guarantees that reserve state and timestamps are always kept in sync with real-time data, preserving integrity across all protocol states.

Customer's response: Fixed in 53f60a.

Fix Review: Fix confirmed.

H-02 `swapMirror()` allows frontrunning to steal user funds by specifying arbitrary sender and recipient

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: PairPoolManager.sol#L541	Status: Fixed	

Description: The `swapMirror()` function in `PairPoolManager` allows the caller to specify both the `sender` (who provides funds for the swap) and the `recipient` (who receives the output). If a user has previously approved the `PairPoolManager` to spend their tokens, an attacker can frontrun their transaction by submitting their own `swapMirror()` call first, setting the victim as `sender` and themselves as `recipient`. This allows the attacker to use the victim's allowance to perform a swap, effectively stealing their funds.

This attack is viable because it's common for users to first approve token allowances in one transaction and then submit the actual swap (or similar function) in a separate transaction. This common pattern increases the likelihood of exploitation via frontrunning.

Exploit Scenario:

1. Alice approves 1000 USDC to the PairPoolManager in transaction A.
2. Alice builds and signs a transaction B calling `swapMirror(sender = Alice, recipient = Alice)`.
3. Bob observes Alice's pending transaction B in the mempool and frontruns it with his own transaction calling `swapMirror(sender = Alice, recipient = Bob)` with similar parameters.
4. Bob receives the swap output using Alice's funds before Alice's transaction executes.

Recommendations: Restrict the sender in `swapMirror()` to be `msg.sender` or make the function to only be callable by the `MarginRouter`. This will prevent unauthorized usage of allowances and protect users from frontrunning attacks.

Customer's response: Fixed in d82d.

Fix Review: Fix confirmed.

H-03 Missing slippage protection in `swapMirror()`

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: PairPoolManager.sol#L522-L529	Status: Fixed	

Description: The `swapMirror` function allows users to swap between two currencies, receiving **lending pool token shares** of the output asset instead of the raw token. The function mirrors the standard swap logic used elsewhere in the protocol but **does not support slippage limits or minimum output constraints**.

This omission introduces significant risk: **users may receive far fewer tokens than expected**, especially under volatile conditions or with unfavorable fee curves. The risk is further amplified by the protocol's **dynamic fee model**, which increases swap fees based on pool imbalance or usage. This can cause substantial losses for users, even in normal conditions without external manipulation.

Additionally, the absence of slippage protection opens the door to **sandwich attacks**. Arbitrageurs can detect incoming `swapMirror` transactions in the mempool and place trades before and after them, amplifying price movements and extracting profit at the user's expense—particularly when dynamic fees are high.

Recommendations: Add a `minAmountOut` parameter to the `swapMirror` function and require the `amountOut` to be greater or equal to it.

Customer's response: Fixed in d82d.

Fix Review: Fix confirmed.

H-04 `removeLiquidity()` is susceptible to sandwich attacks due to missing slippage protection

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: PairPoolManager.sol#L234-L274	Status: Fixed	

Description: The `removeLiquidity()` function allows users to burn LP tokens and withdraw the underlying assets by specifying only the liquidity amount, pool ID, level, and deadline. However, it does not allow users to specify minimum expected token outputs (`minAmount0`, `minAmount1`) as slippage protection.

This omission exposes users to sandwich attacks, where a malicious actor manipulates the pool reserves right before and after the `removeLiquidity()` execution. As a result, users may receive significantly fewer tokens than expected.

Slippage protection is a common and essential defense in DEX designs (e.g., Uniswap) to mitigate such MEV risks.

Recommendations: Extend `RemoveLiquidityParams` to include `minAmount0` and `minAmount1` fields. Enforce these bounds during execution to ensure users can revert if the pool state changes unfavorably before their transaction is mined.

Customer's response: Fixed in d82d.

Fix Review: Fix confirmed.

H-05 Liquidity removals and transfers can be griefed

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: ERC6909Liquidity.sol	Status: Fixed	

Description: The `ERC6909Liquidity` contract enforces a restriction that prevents users from transferring or burning liquidity tokens if less than 30 seconds have passed since their last mint. This restriction applies globally per address and level ID. As a result, an attacker can grief any user by minting a small amount (e.g., 1 wei) of liquidity to a victim's address. This will prevent the victim from transferring or burning any of their liquidity tokens for that level ID, including tokens they legitimately own.

In other words, a malicious user can permanently block a victim's ability to remove liquidity or move their liquidity tokens simply by frontrunning the action and minting to their address. Since the timestamp check is based on `datetimeStore[sender][levelId]` and victims cannot control when others mint to them, this becomes a very cheap griefing vector with no recovery path.

Exploit Scenario:

1. Alice sends a transaction A to transfer her liquidity tokens to someone else or remove liquidity from the pool
2. Bob monitors the mempool, sees Alice's transaction A and frontruns it with a transaction A' which mints a liquidity of 1 wei.
3. Since less than 30 seconds have passed since Bob minted to her, Alice is blocked from calling `transfer()`, `transferFrom()`, or `removeLiquidity()` which causes her transaction A to revert.

Even after 30 seconds, Bob can keep re-minting to Alice, repeatedly resetting the timer and keeping her locked.

Recommendations: Consider removing or redesigning the time lock if it cannot be implemented safely. For example, make sure `datetimeStore` in `addLiquidity()` gets only updated if `receiver == msg.sender` to prevent malicious actors from griefing users.

Customer's response: Fixed in db6c5.

Fix Review: Fix confirmed..

H-06 `MarginRouter::exactOutput()` is vulnerable to sandwich attacks

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: MarginRouter.sol#L115-L124	Status: **Fixed**	

Description: `MarginRouter` is used to initiate and execute swaps using `Uniswapv4::PoolManager`. The contract has two functions `exactInput` and `exactOutput` and both accepts `SwapParams` struct:

```
Unset
struct SwapParams {
        PoolId poolId;
        bool zeroForOne;
        address to;
        uint256 amountIn;
        uint256 amountOutMin;
        uint256 amountOut;
        uint256 deadline;
}
```

While `exactInput` swaps include slippage protection via `amountOutMin`, the `exactOutput` path **lacks an upper bound check on `amountIn`**, which poses a serious risk when users pre-approve large token amounts.

In this scenario, **an attacker can front-run and manipulate the price**, forcing the user to spend significantly more than expected to fulfill the fixed output requirement (`amountOut`). This opens the door for **sandwich attacks** and **excessive token drain**.

Recommendations: Introduce a `maxAmountIn` field to the `SwapParams` struct and enforce this check in the `exactOutput` path. This mirrors best practices from other DEXs (e.g., UniswapV3) and ensures both swap directions are protected against slippage and manipulation.

Customer's response: Fixed in ef8e7.

Fix Review: Fix confirmed..

H-07 `MarginPositionManager::modify` uses wrong amount when decreasing `position.marginAmount`

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: MarginPositionManager.sol#L544-L545	Status: **Fixed**	

Description: When a user opens a margin position, `MarginPositionManager` initiates a `MarginPosition` struct, which has the fields `marginAmount` and `marginTotal` :

```
Unset
MarginPosition memory _position = MarginPosition({
        poolId: params.poolId,
        marginForOne: params.marginForOne,
        marginAmount: params.marginAmount.toUint112(),
        marginTotal: paramsVo.marginTotal.toUint112(),
        borrowAmount: params.borrowAmount.toUint112(),
        rawBorrowAmount: params.borrowAmount.toUint112(),
        rateCumulativeLast: rateCumulativeLast
});
```

Both fields hold an amount, which represents `LendingPoolManager` share (originalAmount), instead of the real token amount. This is the place in the `margin` flow where we convert real token amounts into lending pool share representation (originalAmount).

There is a problem in the `MarginPositionManager::modify` flow: when we decrease the position's `marginAmount`, we decrease the "original" `_position.marginAmount` value with the "real" amount, which results in loss of margin funds for the user.

Another problem is that `lendingPoolManager.withdraw` may not have enough funds to satisfy the full `amount`, which will result in transferring less than the amount to the user.

Exploit Scenario: Imagine `accruesRatioX1120f` for WETH, which is the marginCurrency is 1.2 at point of position creation.

- User deposit 10 WETH, which safe 8.3 as `marginAmount` (10 / 1.2 = 8.33)
- User want to withdraw 1 WETH from his position and calls `modify` with amount = -1
- We enter `LendingPoolManager.withdraw` and transfer 1 WETH to user by burning 0.833 originalTokens from MarginPositionManager
- We update position's `marginAmount` to be 7.3, decreasing its underlying value with 1.2 WETH
- Result is that user 0.2 WETH is blocked in LendingPoolManager

Recommendations:

- Convert `amount` to LendingPool shares representation and decrease `_position.marginAmount` with the correct value

Customer's response: Fixed in 91a5b.

Fix Review: Fix confirmed.

H-08 Miscalculated margin limits allow unsafe borrowing

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: MarginLiquidity.sol#L419-L438	Status: Fixed	

Description: The `getMarginReserves()` function is responsible for computing the margin reserves (`marginReserve0`, `marginReserve1`) and the maximum additional amount that can be borrowed via mirrored reserves (`incrementMaxMirror0`, `incrementMaxMirror1`). However, the implementation contains two critical issues that can lead to incorrect accounting and unsafe overborrowing.

The current computation of margin reserves inverts the retain supply variables:

```
JavaScript
marginReserve0 = Math.mulDiv(totalSupply - retainSupply1, status.reserve0(), totalSupply);
marginReserve1 = Math.mulDiv(totalSupply - retainSupply0, status.reserve1(), totalSupply);
```

This logic mistakenly uses `retainSupply1` when computing `marginReserve0` and vice versa. This leads to incorrect reserve partitioning, resulting in over- or underestimation of available margin liquidity. This can result in borrowers gaining access to more liquidity than is safely allowed.

The values `incrementMaxMirror0` and `incrementMaxMirror1` are meant to represent the maximum additional amount of mirrored reserves that can be safely issued. However, the current formula used is incorrect:

```
Unset
incrementMaxMirror0 = Math.mulDiv(canMirrorReserve0 - status.mirrorReserve0, totalSupply,
retainSupply0);
incrementMaxMirror1 = Math.mulDiv(canMirrorReserve1 - status.mirrorReserve1, totalSupply,
retainSupply1);
```

This way the protocol may incorrectly report borrowing capacity even when the margin reserve is already depleted, allowing unsafe overborrowing and increasing the risk of insolvency.

Recommendations: Fix the the margin reserve calculation to use the matching retain supply:

```
Unset
marginReserve0 = Math.mulDiv(totalSupply - retainSupply0, status.reserve0(), totalSupply);
marginReserve1 = Math.mulDiv(totalSupply - retainSupply1, status.reserve1(), totalSupply);
```

Also, enforce a proper cap on mirrored reserves:

```
Unset
if (retainSupply0 > 0) {
      incrementMaxMirror0 = status.mirrorReserve0 >= marginReserve0 ? 0 : marginReserve0 -
status.mirrorReserve0;
} else { incrementMaxMirror0 = type(uint112).max / 2; }
if (retainSupply1 > 0) {
      incrementMaxMirror1 = status.mirrorReserve1 >= marginReserve1 ? 0 : marginReserve1 -
status.mirrorReserve1;
} else {
      incrementMaxMirror1 = type(uint112).max / 2;
}
```

These changes ensure that margin usage stays within safe bounds and that the protocol's core accounting invariants remain intact.

Customer's response: Fixed in db6c5aa.

Fix Review: Fix confirmed.

H-9 Decreasing margin position using modify may withdraw less than intended funds

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: MarginPositionManager.sol#L543-L544	Status: Fixed	

Description: There is a problem in `MarginPositionManager::modify` flow → When we decrease the position `marginAmount`, we always decrease the `_position.marginAmount` value with the `amount` provided. But `lendingPoolManager.withdraw` may not have enough funds to satisfy the full `amount` , which will result in transferring less than `amount` to the user.

```
Unset
            require(amount <= checker.getMaxDecrease(address(pairPoolManager), _status,
_position), "OVER_AMOUNT");
            lendingPoolManager.withdraw(msg.sender, _position.poolId, marginCurrency,
amount);
            _position.marginAmount -= amount.toUint112();
            if (msg.value > 0) transferNative(msg.sender, msg.value);
```

Recommendations: Check if the requested amount is available in `LendingPoolManager` before proceeding.

Customer's response: Fixed in 91a5b.

Fix Review: Fix confirmed.

Medium Severity Issues

M-01 Donating lending pool tokens deflates Interest accrual for other users

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: LendingPoolManager.sol#L156-L157	Status: Fixed	

Description: In the current interest accrual mechanism, the protocol calculates `accruesRatioX112Of[pool]` based on the total `balanceOriginal` tracked within `LendingPoolManager`. However, if a user **donates** their share tokens directly to the pool (i.e., transfers tokens without a corresponding update to protocol accounting), they **inflate the recorded total supply** without adjusting individual account balances.

This creates an imbalance: the **protocol overestimates the principal base** on which interest should be distributed, thereby **diluting returns for other lenders**.

Exploit Scenario:

- Two users deposit 5,000 tokens each into a lending pool.
- The initial `accruesRatioX112Of[pool]` is 1.0.
- After interest accrues, 2,000 tokens are added to the pool, increasing the ratio to **1.2**. Each user can now withdraw 6,000 tokens.
- One user donates their 5,000 share tokens directly to the `LendingPoolManager`, which increases the internal `balanceOriginal[address(this)][id]` to 15,000.
- The next interest accrual of 2,000 tokens increases the ratio by only **0.133** (`2000 / 15000`) instead of **0.2** (`2000 / 10000`).
- As a result, the victim user receives only **6,666 tokens** instead of the correct **7,000**.

This manipulation deflates future interest for other users, **extracting value from them without changing the exploiter's own withdrawal rights**, since the donated tokens are no longer under their ownership.

Recommendations: Prevent direct token transfers to `LendingPoolManager` and don't allow `LendingPoolManager` contract to be `recipient` of `deposit` function

Customer's response: Fixed in f690fe.

Fix Review: Fix confirmed.

M-02 Position transfer can be frontrun and blocked via Denial-of-Service

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: MarginPositionManager.sol#L128-L137	Status: Acknowledged	

Description: The protocol restricts NFT position transfers such that the **recipient must not already hold a position** for the same pool and margin combination. This is enforced in `MarginPositionManager::update`.

However, this logic is vulnerable to **frontrunning attacks**. An attacker can monitor the mempool for a victim's NFT transfer and quickly send a **dust margin position** to the same recipient address. Since the recipient now appears to hold a position for that pool/margin combo, the original transfer will revert.

This vulnerability can have broader consequences when NFT positions are integrated into other protocols, such as collateralized lending systems. In such cases, an attacker could:

- **Frontrun a liquidation process** by sending a dummy position to the liquidator's address.
- **Block transfers** of NFT positions used in cross-protocol systems, effectively freezing them.
- Cause users to lose liquidation windows or incur penalties due to failed transfers.

Recommendations: Consider removing `_marginPositionIds`

Customer's response: Acknowledged.

M-03 Broken ERC6909 compliance due to events emitting incorrect (unscaled) amounts

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: ERC6909Liquidity.sol #L59	Status: Fixed	

Description: The `ERC6909Liquidity` contract maintains internal balances in unscaled units and applies a scaling factor (`accruesRatioX1120f`) to expose user-facing values. However, the mandatory `Transfer` and `Approval` events emit unscaled internal amounts, not the externally visible scaled values returned by `balanceOf()` or accepted as inputs. This breaks ERC6909 compliance and leads to incorrect data in block explorers, indexers and wallet UIs.

Recommendations: Ensure that all emitted events use scaled amounts consistent with the ERC6909 interface and observable behavior from external calls (#L59 ; #L79 ; #L89 ; #L130 ; #L148).

Customer's response: Fixed in f690fe.

Fix Review: Fix confirmed.

M-04 `MarginPositionManager::close` function doesn't check the position is healthy

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: MarginPositionManager.sol#L364-L376	Status: Fixed	

Description: The `MarginPositionManager::close` function does not validate the health of a position before allowing closure. While it includes PnL and margin release logic, it lacks a check to determine whether the position is currently liquidatable.

```
Unset
int256 pnlAmount = int256(releaseTotalReal) - int256(params.releaseAmount);
uint256 profit;
require(pnlMinAmount == 0 || pnlMinAmount <= pnlAmount, "InsufficientOutputReceived");
if (pnlAmount >= 0) {
	profit = uint256(pnlAmount) + releaseMarginReal;
} else {
	if (uint256(-pnlAmount) < releaseMarginReal) {
		profit = releaseMarginReal - uint256(-pnlAmount);
	} else if (uint256(-pnlAmount) < uint256(_position.marginAmount)) {
		releaseMarginReal = uint256(-pnlAmount);
	} else {
		// liquidated
		revert PositionLiquidated();
	}
}
```

Exploit Scenario: This opens a scenario where a user can repay part of a risky position using all their `marginAmount`, even if the position is technically insolvent. By frontrunning a liquidator, the user may avoid liquidation partially and offload risk onto the protocol or liquidators.

Recommendations: Add a requirement in the function to ensure the position is healthy and not liquidatable before proceeding to its closure.

Customer's response: Fixed in 91a5b7.

Fix Review: Fix confirmed.

M-05 Liquidations prioritize caller and protocol profit over pool solvency

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: [MarginPositionManager.sol#L446-L456]	Status: Fixed	

Description: When a margin position is liquidated, part of the margin is distributed as profit to the liquidator and the protocol. This is calculated as a percentage of the total available margin:

```
Unset
(uint24 callerProfitMillion, uint24 protocolProfitMillion) = checker.getProfitMillions();
// ...
uint256 assetsAmount = _position.marginAmount + _position.marginTotal;
// ...
params.releaseAmount = assetsAmount - profit - protocolProfit;
```

However, if the position is **near insolvency**, subtracting the profit amounts from `assetsAmount` may leave **less than what is needed to repay the debt**. This results in a shortfall, which is implicitly absorbed by the lending pool—reducing the share value for LPs.

This isn't necessarily a malicious exploit; it can happen **organically during normal liquidations** when debt and margin values are close. However, it introduces **systemic risk**, particularly during periods of high volatility or cascading liquidations.

Recommendations: Before distributing profits, ensure:

```
Unset
assetsAmount - profit - protocolProfit >= debtAmount
```

If not, prioritize full repayment and reduce or eliminate profit distribution for that liquidation. This change protects protocol solvency and ensures liquidations serve their intended risk-management role.

Customer's response: Fixed in bc2538.

Fix Review: Fix confirmed.

M-06 User is not enforced to provide `msg.value` on native swaps in `MarginRouter`		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: MarginRouter.sol#L66-L67	Status: Fixed	

Description: In the `MarginRouter` contract, users can initiate token swaps using Uniswap v4 Pool Manager by calling either `exactInput` or `exactOutput`, both of which accept a `SwapParams` struct. These functions delegate to the Uniswap Pool Manager and use `handleSwap` as the `unlock` callback.

In `handleSwap`, tokens are transferred based on the swap direction and amounts.

However, there are two key issues when **the input token is native ETH**:

1. **No check on `msg.value` in `exactInput`**: The contract does not verify that `msg.value == amountIn`, which can result in **unexpected leftover ETH** in the contract, or using router's balance for the user swap.
2. **No refund in `exactOutput`**: If a user sends more ETH than required (e.g., to avoid a revert), the excess is **not refunded**, and remains locked in the contract.

Over time, this can lead to meaningful ETH balances accumulating in the contract. A malicious actor can exploit this by executing a zero-value ETH swap and draining the leftover ETH from previous users.

Recommendations: Implement the following safeguards:

- In the `exactInput` path, **validate `msg.value == params.amountIn`** when the input token is ETH.
- In the `exactOutput` path, **refund any unused ETH** to the sender after completing the swap, if `msg.value > amountIn`.

Customer's response: Fixed in 63601.

Fix Review: Fix confirmed.

M-07 Retroactive application of updated protocol Interest Fee leads to unfair distribution

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: MarginFees.sol#L280-L283	Status: Acknowledged	

Description: When a user interacts with a pool (e.g., deposits, borrows, repays), the action begins with a call to `setBalances`, which internally invokes `_updateInterests`. This function calculates the accrued interest since the last update and distributes it among LPs, depositors, and the protocol.

A portion of this interest is allocated to the protocol based on the return value of `getProtocolInterestFeeAmount`, which is derived from a **protocol owner-controlled variable** via `setProtocolInterestFee`.

Unset

```
if (mirrorReserve0 > 0 && rate0CumulativeLast > status.rate0CumulativeLast) {
    uint256 allInterest0 = Math.mulDiv(
        mirrorReserve0 * UQ112x112.Q112, rate0CumulativeLast, status.rate0CumulativeLast
    ) - mirrorReserve0 * UQ112x112.Q112 + interestX112Store0[poolId];
    uint256 protocolInterest = marginFees.getProtocolInterestFeeAmount(allInterest0);
    if (protocolInterest > UQ112x112.Q112) {
       allInterest0 = allInterest0 / UQ112x112.Q112;
       interestStatus.protocolInterest = protocolInterest / UQ112x112.Q112;
       allInterest0 -= interestStatus.protocolInterest;
       uint256 interest0 =
```

```
            Math.mulDiv(allInterest0, interestReserve0, interestReserve0 +
status.lendingReserve0());
        interestStatus.allInterest = allInterest0;
        interestStatus.pairInterest = interest0;
        if (allInterest0 > interest0) {
            interestStatus.lendingInterest = allInterest0 - interest0;
        }
    } else {
        interest0X112 = allInterest0;
    }
}
```

When the protocol owner changes the `protocolInterestFee`, the new value is immediately applied during the next interaction with any pool. However, the accrued interest being calculated includes all interest since the last update—even interest that accumulated before the fee change.

This causes the newly set protocol fee to be applied retroactively to all previously unclaimed interest. This retroactive application leads to unfair interest distribution, effectively over-allocating to the protocol and under-allocating to lenders and LPs for periods prior to the fee change.

Exploit Scenario:

1. The protocol fee is set to 10%.
2. Over 7 days, a pool accrues 1,000 tokens of interest, but no user interacts with it during that time.
3. On day 8, the protocol owner updates the fee to 50%.
4. A user interacts with the pool, triggering `_updateInterests`.
5. The full 1,000 tokens are now processed using the **new 50% fee**, so the protocol receives 500 tokens instead of the expected 100 — **a 5x overcharge**, reducing the share for LPs and depositors.

Recommendations:

- Track and cache the `protocolInterestFee` value used during the last interest accrual per pool.
- Apply this cached value during future `_updateInterests` calls to calculate protocol share, and only use the new fee value for future interest periods, not retroactively.

Customer's response: Acknowledged.

M-08 Unannounced change in `setLiquidationMarginLevel` can trigger unfair liquidations

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: MarginChecker.sol#L49-L52	Status: Acknowledged	

Description: The `MarginChecker` contract includes an owner-controlled variable, `liquidationMarginLevel`, which defines the minimum margin ratio required to avoid liquidation. Its default value is set to 110%, but the protocol owner can change it at any time without delay or restriction.

This creates two key issues:

1. **Abrupt Liquidation Risk**: An owner can increase the `liquidationMarginLevel` (e.g., from 110% to 130%) without warning. This would immediately mark previously healthy positions as undercollateralized, enabling **unfair liquidations** based on a newly applied threshold.
2. **Lack of Transparency**: The setter function does **not emit an event**, making it harder for users and integrators to monitor changes. This increases the likelihood that users will be unexpectedly liquidated due to unseen parameter changes.

Recommendations: Introduce a cooldown-based update mechanism:

- Add a `newLiquidationMarginLevel` variable and `cooldownStartTime`.
- Create a `getLiquidationMarginLevel()` function that:
  - Returns the active `liquidationMarginLevel`.
  - Promotes `liquidationMarginLevel` after a defined cooldown period (e.g., 24–72 hours).

- Update the setter function (`setLiquidationMarginLevel`) to:
  - Set `newLiquidationMarginLevel`.
  - Start/reset the cooldown timer.
  - Emit an event to signal the pending change.

Customer's response: Acknowledged.

M-09 `_getPriceDegree` may wrongly return high fee for small swap

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: MarginFees.sol#L80-L82	Status: Acknowledged	

Description: The protocol uses `_getPriceDegree` to estimate how much a swap or margin operation moves the price. This is done by comparing the real reserves after the swap to the last recorded price, using the following logic:

```
Unset
uint224 price0X112 = UQ112x112.encode(_reserve1.toUint112()).div(_reserve0.toUint112());

uint224 price1X112 = UQ112x112.encode(_reserve0.toUint112()).div(_reserve1.toUint112());

uint256 degree0 = differencePrice(price0X112, lastPrice0X112).mulMillionDiv(lastPrice0X112);

uint256 degree1 = differencePrice(price1X112, lastPrice1X112).mulMillionDiv(lastPrice1X112);

degree = Math.max(degree0, degree1);
```

This approach is wrong because after a big swap (for example moving the price by 20-30%), `truncatedReserves` are not updated in full and consecutive operations may result in paying large swap fees without moving the real price.

However, `lastPrice0X112` and `lastPrice1X112` are based on `truncatedReserves`, which are not immediately or fully updated after a large price-moving trade. This results in stale price references being used to calculate price movement for subsequent operations.

Exploit Scenario: This situation may occur without malicious intent, but it is also **potentially exploitable** by an attacker:

1. A margin position significantly shifts the price (e.g., 20–30%) by borrowing which **does not trigger swap fees**.
2. The `truncatedReserves` used to calculate `lastPriceX112` are not yet updated to reflect this movement.
3. A subsequent **swap price impact**, possibly by another user, is evaluated using stale price data.
4. The protocol overestimates price movement and **applies an inflated dynamic fee**, even if the swap itself didn't affect the price.
5. A malicious actor could deliberately front-run swaps to inflate fees for others, extracting value without bearing the cost of slippage.

Recommendations: Ensure the reserve values used in `_getPriceDegree` reflect the real, current pool state, not `truncatedReserves`.

Customer's response: This behavior is part of a pre-designed anti-arbitrage mechanism. While it may increase the swap fee under certain conditions, it is constrained by slippage limits.We have decided to retain this mechanism.

M-10 Anyone can open position for another user, which increases victim's risk

Severity: **Medium**	Impact: **Medium**	Likelihood: **Low**
Files: MarginPositionManager.sol#L210-L211	Status: Acknowledged	

Description: The protocol allows users to open leveraged positions under a shared margin pool. However, the interest rate is influenced by the aggregate borrowing behavior within that pool. This creates a vulnerability where an attacker can grief another user by opening a high-leverage position just before the victim's transaction is confirmed.

Although only the position owner can modify their own margin position, if a user is about to initialize a new position (e.g., at 2x leverage), an attacker can front-run that transaction and open a larger, riskier position (e.g., 5x leverage) on the same pool and margin option. This malicious position will:

- Increase interest owned by the victim
- Risk of liquidations

Recommendations: Disallow minting leveraged positions (leverage > 1x) on behalf of other users.

Customer's response: When recipients operate independently, they should monitor their position status and thus avoid personal losses.

M-11 Improper Interest accrual on re-enabling Interest for pool

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: PoolStatusManager.sol #L460-L461	Status: Partially fixed	

Description: When a pool is initialized, `blockTimestampLast` is set to `block.timestamp` and later used in `_updateInterest` to calculate accrued interest. The protocol allows interest accrual to be toggled off via the `setInterestClosed` function, making the pool temporarily interest-free.

However, if interest is re-enabled after being off for a period of time, the **previous `blockTimestampLast` is reused**, causing interest to accrue over the entire disabled duration. This can result in **unexpectedly large interest charges** for users who borrow just before interest is re-enabled.

Recommendations: Update the logic in `setInterestClosed` or `setBalances` to reset `blockTimestampLast` to the current block timestamp when interest is re-enabled. This ensures that interest only accrues from the moment interest is turned back on, preserving fairness and expected behavior.

Customer's response: Partially fixed in 4ed9d3.

Fix Review: The current fix doesn't cover a scenario, where the contract owner disables interest for the pool. No interest will be accrued between the user's last interaction with the pool and the transaction that disables it.

Low Severity Issues

L-01 `mirrorTokenManager` can add new managers but can't revoke old ones

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: MirrorTokenManager.sol#L71-L79	Status: Acknowledged	

Description: In the `MirrorTokenManager` contract, the owner can add new pool managers using the `addPoolManager` function, which designates up to three new addresses as valid `poolManagers`. However, **there is no mechanism to revoke or remove previously added managers**.

This results in **permanent minting authority** for any address ever assigned as a manager, even if that access is no longer appropriate. The same issue exists in the `MarginLiquidity` contract (#L255-L261), where manager roles cannot be revoked once granted.

Recommendations: Implement a function responsible for removing managers.

Customer's response: Acknowledged.

L-02 `estimatePNL` always returns wrong value for borrow positions

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: MarginChecker.sol#L147-L149	Status: Fixed	

Description: The `estimatePNL` function produces incorrect results for positions with no leverage—specifically, those where `position.marginTotal == 0`. In such cases, the calculation:

```
Unset
uint256 releaseTotal = uint256(_position.marginTotal).mulDivMillion(closeMillionth);
pnlAmount = int256(releaseTotal) - int256(releaseAmount);
```

will always evaluate to:

```
Unset
pnlAmount = 0 - releaseAmount = -releaseAmount
```

This leads to a consistently negative PnL, regardless of the actual market performance or borrow repayment.

- The incorrect PnL value is not used in protocol-critical logic.
- Currently, it only affects event emissions and view functions, such as external analytics or UI components.

- However, this could cause misleading user interfaces, incorrect dashboards, or confusion for integrators relying on this output.

Recommendations: Add a conditional branch in `estimatePNL` to handle cases where `marginTotal == 0`:

- Either return `pnlAmount = 0` directly, or
- Compute the PnL based solely on borrow repayment logic, if relevant

Customer's response: Partially fixed in 26213.

Fix Review: The described issue is fixed. However, waiting for customer to confirm another issue

L-03 `interestOperator` mapping becomes outdated if `statusManager` is changed in the pool manager

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: MarginLiquidity.sol#L259	Status: Acknowledged	

Description: The `interestOperator` mapping in `MarginLiquidity` is updated when a `poolManager` is added via `addPoolManager()`, which retrieves the current `statusManager` from the `poolManager` and marks it as authorized. However, if the `poolManager` subsequently changes its `statusManager` using `setStatusManager()`, this new address does not get automatically whitelisted in `MarginLiquidity`. As a result, the new `statusManager` will be unauthorized to call functions like `addInterests()` that are gated by the `onlyInterestOperator` modifier. This introduces an operational risk: unless the owner re-calls `addPoolManager()` to re-sync the new `statusManager`, interest updates may halt unexpectedly.

Recommendations: Consider designing the system to dynamically query the current `statusManager` from the `poolManager` instead of relying on a static `interestOperator` mapping.

Alternatively, make the dependency and required update process more explicit or automate the sync of `interestOperator` during `statusManager` updates in `PairPoolManager`.

Customer's response: Acknowledged.

Fix Review: Acknowledged.

L-04 Hardcoded global slippage tolerance (`maxSliding`) limits user control and may cause liquidity provisioning failures

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: PairPoolManager.sol#L210-L211	Status: Fixed	

Description: The protocol applies a hardcoded global slippage tolerance (`maxSliding`, defaulting to 0.5%) to the `addLiquidity()` logic. This value is enforced across all pools during liquidity provisioning via the `computeLiquidity()` function, which ensures the added liquidity amounts `amount0` and `amount1` do not deviate beyond the `maxSliding` threshold.

While this provides baseline protection against adverse price movement, it presents two key issues:

1. **Users cannot specify slippage tolerance** — Unlike most major DEXes (e.g., Uniswap), users have no control over slippage. They are forced to accept the global protocol-defined threshold, which may not align with their risk preferences.
2. **One-size-fits-all logic across all pools** — The same maxSliding value is used across all pools, regardless of liquidity depth or volatility. As a result:
    - Pools with low liquidity or high volatility may revert liquidity operations unnecessarily.
    - Users may face usability issues during market turbulence, even when their desired slippage range is acceptable to them.

While not directly exploitable, this rigidity degrades protocol composability and makes integrations and advanced user workflows more fragile.

Recommendations: Allow users to specify their desired slippage tolerance directly via `addLiquidity()` parameters. Alternatively, consider implementing pool-specific `maxSliding` values that can be dynamically adjusted based on liquidity conditions or volatility metrics.

Customer's response: Fixed in 16ea6.

Fix Review: Fix confirmed.

Informational Issues

I-01. Consider implementing a multihop swap mechanic in `MarginRouter`

Description: Uniswap V4 unlocks truly gas-efficient multi-hop swaps by returning only net token deltas.

Recommendation: Consider integrating delta-tracking mechanism into the `MarginRouter` to minimize per-hop transfer and approval overhead.

Customer's response: Acknowledged.

I-02. Consider emitting events on important state changes

Description: Consider emitting events when owner setters are executed setting important state variables to new values.

For example in `MarginFees` :

- setRateStatus
- setProtocolInterestFee
- setProtocolMarginFee
- setProtocolSwapFee
- setMarginFee

Customer's response: Fixed in 2fdfc8.
Fix Review: Fix confirmed.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

The review was conducted over a limited timeframe and may not have uncovered all relevant issues or vulnerabilities.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.