LIKWID Due Diligence Report

Please Note

1. The analysis of the Severity is purely based on the smart contracts mentioned in the Audit Scope and does not include any other potential contracts deployed by the Owner. No applications or operations were reviewed for Severity. No product code has been reviewed.

2. Due to the time limit, the audit team did not do much in-depth research on the business logic of the project. It is more about discovering issues in the smart contracts themselves.

Audit Period: 2025/4/05 - 2025/04/09 (YYYY/MM/DD)

Based on these links:

Item	Description
Project Name	LIKWID
Website	https://likwid.fi/
Twitter	https://x.com/likwid_fi
Documentation	https://likwidfi.gitbook.io/likwid-protocol-docs
Github	https://github.com/likwid-fi/likwid-margin/tree/v2
External Audit Reports	-

Overall Risk: Low

Likwid Finance is a decentralized finance (DeFi) project whose core product is a "fully permissionless, oracle-free margin trading protocol" based on Uniswap V4. The project aims to provide innovative financial tools, which may include features such as liquidity mining and margin trading. The project is currently in the testing phase.

- External Partnerships
  Partnerships Mentioned in Tweets: Tweets from @likwid_fi explicitly mention collaborations or endorsements with Uniswap Foundation, BNB Chain, and APR Labs, indicating that the project has established multiple partnerships within the DeFi ecosystem.

    - Association with @UniswapFND (April 1, 2025): The project was recognized as a "winner" by the Uniswap Foundation, which may indicate that its technology or innovation has received initial validation from a well-known institution.
    - Association with @BNBCHAIN (April 4, 2025): Likwid Finance was selected by BNB Chain as one of the 16 early-stage projects for the 9th season of the "Most Valuable Builder (MVB)" accelerator program.
    - Collaboration with @apr_labs (April 2, 2025): @likwid_fi announced a partnership with @apr_labs to launch a "Dual Rewards & anti-MEV Model," involving the MON/aprMON liquidity mining program. This indicates a clear collaboration with APR Labs to jointly develop features.

- No audit report
  The project currently does not have a publicly available security audit report, which may indicate that this aspect is either undisclosed or incomplete.
  Without a thorough audit, the smart contract may contain undiscovered vulnerabilities or bugs. These weaknesses can be exploited by malicious actors, leading to potential hacks, theft of funds, or other security breaches.

- Centralization Risks in Smart Contracts
  We conducted a security check on the project's smart contracts and identified several issues across multiple contracts, including excessive privileges of the privileged role, insufficient input validation, lack of event logging for critical operations, and observation initialization issues. We recommend mitigating these risks by following best security practices for on-chain parameter configuration to reduce centralization risks.

Therefore, we rate the overall risk level as **Low**.

Smart Contracts Risks

No.	Contract Name	Type	Contract	Verdict	Details
1	LendingPoolManager	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/LendingPoolManager.sol	Green	
2	MarginPositionManager	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginPositionManager.sol	Green	
3	PairPoolManager	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/PairPoolManager.sol	Green	
4	MarginChecker	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginChecker.sol	Low	[L01]
5	PoolStatusManager	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/PoolStatusManager.sol	Low	[L02] [L03]
6	MarginFees	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginFees.sol	Low	[L02] [L03] [L04]
7	MarginLiquidity	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginLiquidity.sol	Low	[L02] [L03] [L04]
8	MarginOracle	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginOracle.sol	Low	[L05]

9	MarginRouter	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginRouter.sol	Green	
10	MarginHook	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MarginHook.sol	Green	
11	MirrorTokenManager	Business	https://github.com/likwid-fi/likwid-margin/blob/v2/src/MirrorTokenManager.sol	Green	

Risk Details

[L01] Excessive Privileges of the Privileged Role `owner`	
Contract	MarginChecker
Severity Level	Low
Description	In the smart contract MarginChecker, the privileged role (i.e., the contract owner) has significant authority to perform several critical operations, such as setting caller profit, protocol profit, liquidation margin level, minimum margin level, and minimum borrow level. These operations directly impact the contract's functionality and the security of users' funds. If these privileges are abused or maliciously exploited, it could lead to the following risks: ● Malicious Parameter Setting: The contract owner can arbitrarily change key parameters like callerProfit, protocolProfit, liquidationMarginLevel, etc., potentially leading to unfair profit distribution or unreasonable liquidation conditions, harming users' interests. ● Single Point of Failure: If the contract owner's private key is compromised or lost, an attacker could use these privileges to perform malicious operations, resulting in user fund losses. ● Lack of Transparency: Frequent changes to key parameters may decrease users' trust in the contract, affecting its usage and adoption.
Recommendation	To mitigate the risks associated with excessive privileges of the privileged role, the following measures are recommended:

	• Multisignature Wallet: Transfer the contract owner's privileges to a multisignature wallet, ensuring that critical operations require the joint signatures of multiple trusted parties. This can effectively prevent single point of failure and malicious operations. • Timelock Mechanism: Introduce a timelock mechanism to ensure that changes to key parameters require a certain delay period before taking effect, giving users sufficient time to understand and respond to these changes. • Transparency and Auditing: Add event logging in the contract to ensure that all critical operations have corresponding event records, facilitating auditing and monitoring. Additionally, conduct regular third-party security audits to ensure the contract's security and reliability. By implementing these measures, the risks associated with excessive privileges of the privileged role can be effectively mitigated, enhancing the contract's security and users' trust.

[L02] Excessive Owner Privileges	
Contract	PoolStatusManager
Severity Level	**Low**
Description	In the provided smart contract PoolStatusManager, the owner has significant privileges and can perform several critical operations, such as setting fee status, setting margin fees, and setting margin oracles. These operations directly impact the contract's functionality and the security of users' funds. If these privileges are abused or maliciously exploited, it could lead to the following risks: • Malicious Parameter Setting: The contract owner can arbitrarily change critical parameters, such as marginFees and marginOracle, which may result in unfair fee distribution or unreasonable oracle data, harming users' interests. • Single Point of Failure: If the contract owner's private key is compromised or lost, an attacker could use these privileges to perform malicious operations, leading to user fund losses. • Lack of Transparency: Frequent changes to critical parameters may lead to a decrease in user trust in the contract, affecting its usage and adoption. The MarginFees and MarginLiquidity contract also has similar risks.
Recommendation	To mitigate the risks associated with excessive owner privileges, the following measures are recommended: • Multisignature Wallet: Transfer the contract owner's privileges to a multisignature wallet, ensuring that critical operations require the joint signature of multiple trusted parties. This can effectively prevent single points of failure and malicious operations.

	• Timelock Mechanism: Introduce a timelock mechanism to ensure that changes to critical parameters require a certain delay before taking effect, giving users enough time to understand and respond to these changes. • Transparency and Auditing: Add event logging to the contract to ensure that all critical operations have corresponding event records for auditing and monitoring. Additionally, conduct regular third-party security audits to ensure the contract's security and reliability. By implementing these measures, the risks associated with excessive owner privileges can be effectively reduced, enhancing the contract's security and user trust.

[L03] Insufficient Input Validation	
Contract	PoolStatusManager
Severity Level	**Low**
Description	In the provided smart contract PoolStatusManager, certain functions lack sufficient validation of input parameters. This can lead to unexpected behavior or security vulnerabilities, with specific risks as follows: • Invalid Address: Functions such as setMarginFees and setMarginOracle do not validate the input addresses. If an invalid address (e.g., zero address) is passed, it may cause the contract to malfunction or trigger other security issues. • Invalid Amount: Functions such as getAmountOut and getAmountIn do not sufficiently validate the input amounts. If an invalid or unreasonable amount is passed, it may result in calculation errors or other unexpected behavior. • Boundary Conditions: Lack of boundary condition checks for input parameters may lead to overflow, underflow, or other unexpected behavior, affecting the contract's security and stability. The MarginFees and MarginLiquidity contracts also has similar risks.

Recommendation	To mitigate the risks associated with insufficient input validation, the following measures are recommended:
	• Address Validation: Add validation for address inputs in functions to ensure that the passed address is not a zero address or other invalid address.
	JavaScript
	`function setMarginFees(address _marginFees) external onlyOwner {`
	`  require(_marginFees != address(0), "Invalid address");`

```
    marginFees = IMarginFees(_marginFees);
}
```

- Amount Validation: Add validation for amount inputs in functions to ensure that the passed amount is within a reasonable range.

```JavaScript
function getAmountOut(uint256 amountIn) external view
returns (uint256) {
    require(amountIn > 0, "Amount must be greater than
zero");
    // Additional logic
}
```

- Boundary Condition Checks: Add boundary condition checks for all functions involving input parameters to ensure that the input parameters are within expected ranges, preventing overflow and underflow issues.

```JavaScript
function setFeeStatus(PoolId poolId, uint24 _marginFee)
external onlyOwner {
    require(_marginFee >= 0 && _marginFee <= 10000,
"Invalid fee");
    statusStore[poolId].marginFee = _marginFee;
}
```

- Event Logging: Ensure that all critical operations have corresponding event records for auditing and monitoring.

```JavaScript
event MarginFeesSet(address marginFees);

function setMarginFees(address _marginFees) external
onlyOwner {
```

```
    require(_marginFees != address(0), "Invalid
address");
    marginFees = IMarginFees(_marginFees);
    emit MarginFeesSet(_marginFees);
}
```

By implementing these measures, the risks associated with insufficient input validation can be effectively reduced, enhancing the contract's security and reliability.

[L04] Event Logging for Critical Operations	
Contract	MarginFees
Severity Level	Low
Description	The MarginFees contract lacks event logging for critical operations. This absence can lead to insufficient transparency and difficulty in tracking key operations during audits. The MarginLiquidity contract also has similar risks.

Recommendation	It is recommended to add event logging for critical operations to enhance transparency and facilitate auditing.
	
	Below are the suggested events example and corresponding function modifications:
	
	JavaScript
	`event FeeToUpdated(address indexed newFeeTo);`
	`event ProtocolFeeUpdated(uint24 newProtocolFee);`
	`event RateStatusUpdated(RateStatus newRateStatus);`
	`event DynamicFeeMinDegreeUpdated(uint24 newDynamicFeeMinDegree);`
	`event DynamicFeeUnitUpdated(uint24 newDynamicFeeUnit);`
	
	`function setFeeTo(address _feeTo) external onlyOwner {`
	`    require(_feeTo != address(0), "Invalid address");`
	`    feeTo = _feeTo;`
	`    emit FeeToUpdated(_feeTo);`

```
}

function setProtocolFee(uint24 _protocolFee) external
onlyOwner {
    protocolFee = _protocolFee;
    emit ProtocolFeeUpdated(_protocolFee);
}

function setRateStatus(RateStatus calldata _status)
external onlyOwner {
    rateStatus = _status;
    emit RateStatusUpdated(_status);
}

function setDynamicFeeMinDegree(uint24
_dynamicFeeMinDegree) external onlyOwner {
    dynamicFeeMinDegree = _dynamicFeeMinDegree;
    emit
DynamicFeeMinDegreeUpdated(_dynamicFeeMinDegree);
}

function setDynamicFeeUnit(uint24 _dynamicFeeUnit)
external onlyOwner {
    dynamicFeeUnit = _dynamicFeeUnit;
    emit DynamicFeeUnitUpdated(_dynamicFeeUnit);
}
```

By adding these events, the transparency of contract operations can be significantly improved, and valuable operation records can be provided during audits.

[L05] Observation Initialization Issue	
Contract	MarginOracle
Severity Level	Low

Description	In the MarginOracle contract, the write function is responsible for initializing observation data. However, there is a potential risk in the current implementation where it does not check if reserve0 and reserve1 are non-zero during initialization. If these reserve values are zero at the time of initialization, it could lead to incorrect initialization, thereby affecting subsequent observation data and the normal operation of the contract. Here is the current code snippet: JavaScript `if (_state.cardinality == 0) { (_state.cardinality, _state.cardinalityNext) = observations[hook][id].initialize(_blockTimestamp(), reserve0, reserve1); }` If reserve0 and reserve1 are zero during initialization, the observation data will be initialized based on zero values. This could lead to inaccurate subsequent observation data, affecting the normal operation and reliability of the contract. Data Integrity Issues: Incorrect data at initialization will impact all subsequent operations and decisions based on these observations, potentially leading to incorrect business logic execution and data analysis results. Potential Security Vulnerability: An attacker could exploit this by providing zero values during initialization to corrupt the integrity of the observation data, thereby affecting the normal operation of the contract.
Recommendation	To avoid the above risks, it is recommended to add checks during initialization to ensure that reserve0 and reserve1 are non-zero. Here is an improved code example: JavaScript `if (_state.cardinality == 0) { require(reserve0 > 0 && reserve1 > 0, "Reserves must be non-zero for initialization");`

```
    (_state.cardinality, _state.cardinalityNext) =

observations[hook][id].initialize(_blockTimestamp(),
reserve0, reserve1);
}
```

By adding the above checks, we can ensure that reserve0 and reserve1 are non-zero during initialization, thereby avoiding incorrect initialization and ensuring the accuracy of observation data and the normal operation of the contract.