CYBRO × PESSIMISTIC

# SECURITY ANALYSIS

by Pessimistic

This report is public

**October 6, 2025**

# ABSTRACT

This report evaluates the security of the smart contracts in the CYBRO LPManager project. The objective was to identify and document security vulnerabilities within the platform's smart contracts.

# DISCLAIMER

This audit makes no warranties or guarantees as to the security of the code. A single audit is insufficient to ensure complete security. We strongly recommend conducting multiple independent audits and implementing a public bug bounty program to enhance smart contract security. Additionally, this security audit should not be construed as investment advice.

# SUMMARY

This report assesses the security of the CYBRO LPManager smart contracts. The audit process is detailed in the following section.

The audit identified several medium severity issues: No documentation, Discrepancy with NatSpec comments, The owner's role, and Overflow. Also, several low severity issues were found.

Overall code quality is good.

After the initial audit, the codebase was updated. The No documentation issue of medium severity was addressed, and the Discrepancy with NatSpec comments issue of medium severity was fixed. The owner's role and Overflow issues of medium severity were commented on by the developers. All low severity issues and notes were either fixed or commented on. The test passed.

# GENERAL RECOMMENDATIONS

We do not have any further recommendations.

# PROJECT OVERVIEW

## Project description

For this audit, we reviewed the CYBRO LPManager project from a private GitHub repository, commit c7df4976c0e53a22a15179ebece4cead4511f63e.

The scope of the audit included:

- **LPManager.sol**;
- **ProtocolFeeCollector.sol**;
- **interfaces/ILPManager.sol**;
- **interfaces/IProtocolFeeCollector.sol**.

The list of networks for deployment:

- Arbitrum;
- Base;
- Unichain.

The project has only 1 test, that passes successfully. The code coverage is 87.93%.

The total lines of code (LOC) of audited sources is 680.

## Codebase update #1

After the initial audit, the codebase was updated. For the recheck, we were provided with commit b13d407d9271a8936e51e27e9c45a1dd7c43b5b2.

The 1/1 test passed, the code coverage was 86.74%.

All issues were fixed or commented on by the developers.

# AUDIT PROCESS

The audit was conducted from September 25 to September 30, 2025.

We began by reviewing the provided materials. Then we performed a comprehensive manual review of all contracts within the audit scope, verifying the logic and security properties, including:

- Standard Solidity issues;
- Integration with Uniswap V3;
- Fee calculation and charging logic;
- Calculation of the ratio between token0 and token1 for depositing to Uniswap V3.

We analyzed the project with the following automated tools:

- AI scanner AuditAgent;
- AI scanner Savant Chat;
- Static analyzer Slither;
- Our plugin Slitherin with an extended set of rules;
- Semgrep rules for smart contracts.

We executed the test suite and calculated code coverage.

All identified issues — whether detected manually or through automated tools — were compiled into the private report.

We performed the recheck on October 6, 2025. We checked whether the previously reported issues were fixed, re-ran the tests, and recalculated the code coverage.

Finally, we updated the report.

# MANUAL ANALYSIS

All contracts were manually inspected to validate their logic and security. Additionally, automated analysis results were manually verified. Confirmed issues are documented below.

## Critical issues

Critical issues pose severe risks to project security, potentially leading to fund loss or other catastrophic failures. Contracts should not be deployed until such issues are resolved.

**No critical issues were identified during the audit.**

## Medium severity issues

Medium severity issues may impact the project's current functionality. This category includes bugs, potential revenue loss, operational inefficiencies, and risks related to improper system management. We strongly recommend addressing these issues.

### M01. No documentation (addressed)

The project has no documentation. Documentation is a critical part that helps to improve security and reduce risks. It should explicitly describe the purpose and behavior of the contracts, their interactions, and key design choices. Documentation is also essential for any further integrations.

As an example, due to the lack of documentation, it was unclear how certain fees are intended to be applied. Specifically, the `LPManager.moveRange` function charges `liquidityProtocolFee` on the entire position (initial principal + claimed fees). In contrast, the `compoundFees` function charges the fee only on accrued rewards.

> The developers added more detailed NatSpec comments.

### M02. Discrepancy with NatSpec comments (fixed)

The `LPManager.createPosition` function charges `liquidityProtocolFee`. However, according to the NatSpec comment at line 264, it should charge the `depositProtocolFee` instead.

> The issue has been fixed and is not present in the latest version of the code.

### M03. The owner's role (commented)

In the current implementation, the system depends heavily on the owner role. Thus, there are scenarios that can lead to undesirable consequences for the project and its users, e.g., if owner's private keys become compromised.

In the **ProtocolFeeCollector** contract, the owner can set different types of fees through the `setFees`, `setLiquititityProtocolFee`, `setFeesProtocolFee`, `setDepositProtocolFee` functions. However, these functions do not enforce an upper bound on the fee values. As a result, the owner could configure fee values greater than 100%, which would effectively block the functionality of the **LPManager** contract.

In addition, setting fees close to 100% would allow the owner to capture nearly all user assets. In the worst case, the owner could front-run a large deposit by setting the high fees and draining user funds.

We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig.

> `Comment from the developers:`
>
> The problem is known and will not be fixed, as the contract owner is a multisig wallet. Deployed contracts:
>
> - Unichain 0xfB84664E669f8aaE284C626dC7b31c4e45101348;
> - Arbitrum 0xfD3BE1d641aDAa64C005dC2D6A2Ff5bEA96AD744;
> - Base 0x567103a40C408B2B8f766016C57A092A180397a1.

### M04. Overflow (commented)

In the `LPManager._toOptimalRatio` function, the expression: `uint256(prices.current) * uint256(prices.current)` may overflow and cause a revert. Since `prices.current` has `sqrtPriceX96` type.

This could make it impossible to open new positions or increase liquidity in existing ones for certain price ranges, effectively limiting protocol functionality.

> `Comment from the developers:`
>
> Acknowledged, we won't fix.

## Low severity issues

Low severity issues do not immediately affect project operations but may introduce risks in future iterations. We recommend resolving these issues or providing justification for the chosen implementation.

### L01. Duplicate fee logic across contracts (fixed)

Fee calculation (in the `calculateLiquidityProtocolFee`, `calculateFeesProtocolFee`, `calculateDepositProtocolFee` functions of the **ProtocolFeeCollector** contract) and collection logic (in the `_collectFeesProtocolFee`, `_collectLiquidityProtocolFee`, `_collectDepositProtocolFee` functions of the **LPManager** contract) is duplicated across multiple functions for different fee types (deposit, fees, liquidity). This increases code size and reduces readability. Consider creating a single function in each contract (one private in the **LPManager** contract, one external in the **ProtocolFeeCollector** contract) and passing the fee type as a parameter.

> The issue has been fixed and is not present in the latest version of the code.

### L02. Gas consumption (fixed)

The `ProtocolFeeCollector.withdrawProtocolFees` function accepts a single token address per call. If the protocol collects fees in multiple tokens, the owner must call the function separately for each token. Consider adding functionality to withdraw fees for multiple tokens in a single transaction to reduce gas consumption.

> The issue has been fixed and is not present in the latest version of the code.

### L03. Missing events (fixed)

In the **ProtocolFeeCollector** contract, the constructor does not emit events when setting `liquidityProtocolFee`, `feesProtocolFee`, and `depositProtocolFee` variables. This would allow easier tracking and auditing of the initial fee setup.

> The issue has been fixed and is not present in the latest version of the code.

### L04. Typo in the function name (fixed)

The function name `ProtocolFeeCollector.setLiquitityProtocolFee` contains a typo. The correct spelling should be `setLiquidityProtocolFee`. The same typo exists in the **IProtocolFeeCollector** interface.

> The issue has been fixed and is not present in the latest version of the code.

### L05. Unused import (fixed)

The **LPManager** contract has unused import of the **LiquidityAmounts** contract at line 12.

> The issue has been fixed and is not present in the latest version of the code.

### L06. Using Ownable2Step (commented)

The **ProtocolFeeCollector** contract inherits from the standard **Ownable** contract. Ownership can be transferred in a single step using `transferOwnership` function. However, single-step transfers carry a small risk of accidental or malicious ownership change if transferOwnership is called incorrectly. To mitigate this risk, consider using Ownable2Step for a safer two-step ownership transfer process.

> `Comment from the developers:`
>
> Acknowledged, we won't fix.

### L07. Zero price (commented)

In the `LPManager._getPriceFromPool` function, if the rate of `token1` relative to `token0` is less than 1, the numerator at line 894 may be smaller than the denominator, causing the calculated `price` to be 0. Consider multiplying the `ratio` at line 893 by the decimals of `token1` to preserve precision and avoid a zero price.

> `Comment from the developers:`
>
> Acknowledged, we won't fix.

## Notes

### N01. Direct Uniswap calls bypass LPManager logic (commented)

Users can bypass the **LPManager** contract by interacting directly with Uniswap V3 position manager. In such cases, the internal logic of the **LPManager** contract may become inconsistent with the actual Uniswap position state. For example, if a user calls the `decreaseLiquidity` function directly on Uniswap:

- The withdrawn amount (principal) is not returned immediately, but recorded in Uniswap's internal `tokensOwed` mapping;
- These funds can then be collected via the `collect` function;
- When the user later calls the `compoundFees`, `moveRange`, or `claimFees` function through the **LPManager** contract, the functions may incorrectly treat this amount as accrued fees.

> `Comment from the developers:`
> Acknowledged, we won't fix.

This analysis was performed by Pessimistic:

**Daria Korepanova**, Senior Security Engineer
**Evgeny Bokarev**, Security Engineer
**Irina Vikhareva**, Project Manager
**Alexander Seleznev**, CEO

**October 6, 2025**