CYBRO × PESSIMISTIC

SECURITY ANALYSIS

by Pessimistic

This report is public

July 1, 2025

Abstract 2
Disclaimer 2
Summary 2
General recommendations 2
Project overview 3
Project description 3
Audit process 4
Manual analysis 5
Critical issues 5
C01. Inflation attack (commented) 5
Medium severity issues 6
M01. Insufficient documentation (commented) 6
M02. Management fee calculation (commented) 6
M03. Potential use of stale Chainlink prices (commented) 7
M04. Project's roles (commented) 7
M05. Staked amount can be decreased with previous signatures (commented) 8
Low severity issues 9
L01. Code style (commented) 9
L02. Missing event emission in setter (commented) 9
L03. Missing zero address check for fee recipient (commented) 9
L04. No revert in case of invalid condition (commented) 10
L05. Unused import (commented) 10
L06. Using ERC-7201 (commented) 10
Notes 10
N01. Minimal logic for emergency withdrawal (commented) 10

ABSTRACT

In this report, we consider the security of smart contracts of Cybro LidoVault project. Our task is to find and describe security issues in the smart contracts of the platform.

DISCLAIMER

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

SUMMARY

In this report, we considered the security of Cybro LidoVault smart contracts. We described the audit process in the section below.

The audit showed one critical issue: Inflation attack. The audit also revealed several issues of medium severity: Insufficient documentation, Management fee calculation, Potential use of stale Chainlink prices, Project's roles, Staked amount can be decreased with previous signatures. Moreover, several low-severity issues were found.

After the initial audit, the developers reviewed the report and commented on all the findings.

GENERAL RECOMMENDATIONS

We recommend fixing the mentioned issues.

PROJECT OVERVIEW

Project description

For the audit, we were provided with Cybro LidoVault project on a public GitHub repository, commit 838fb0715917b9619e3465f6932bd0b9370f491e.

The scope of the audit included:

- **src/vaults/LidoVault.sol**;
- **src/BaseVault.sol**;
- **src/FeeProvider.sol**.

The documentation for the project included https://docs.cybro.io/cybro and the management fee calculation formula description.

The project has a total of 44 tests, all of which pass. 2 of these tests are related to the current scope and also pass. The code coverage of the current scope is 83.75%.

The total LOC of audited sources is 545.

AUDIT PROCESS

We started the audit on June 25 and finished on June 30, 2025.

We inspected the materials provided for the audit.

During the work, we stayed in touch with the developers and discussed confusing or suspicious parts of the code.

We manually analyzed all the contracts within the scope of the audit and checked their logic. Among other, we verified the following properties of the contracts:

- The influence of the owner and admin roles on the project;
- Standard Solidity issues;
- Whether it is possible to steal or lock user tokens;
- The correctness of formulas and their correspondence to the documentation;
- The integration with Chainlink and proper usage of its components;
- The correct use of signatures and signature validation logic.

We scanned the project with the following tools:

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

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tools.

On July 1, the report was updated after the developers reviewed it and marked all identified issues as "acknowledged, we won't fix".

MANUAL ANALYSIS

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

C01. Inflation attack (commented)

The `BaseVault.deposit` function is vulnerable to an inflation attack.

An attacker can first deposit 1 `WEI`, setting `totalSupply = totalAssets = 1`. Later, the attacker front-runs the victim's deposit transaction, manipulating the denominator in the share calculation.

For instance, if the victim tries to deposit 1 `WETH`, the attacker deposits a slightly higher amount (e.g., `1.0038 wstETH` or similar, depending on the price) in the same block. The result can be the following:

- If the victim sets `minShares == 0`, the victim receives 0 shares and loses their full deposit.
- If the victim sets `minShares > 0`, the attacker can repeatedly front-run new deposits, causing each transaction to revert due to failing the `require(shares >= minShares)` check. It locks the vault and prevents any new deposits.

> `Comment from the developers:`
>
> **To ensure the safety of our users' deposits, we implement a mitigation strategy known as "dead shares." Before publishing the vault, we always make a small deposit ourselves and are ready to lose it. For the rest of the users depositing will always be safe.**
>
> **While we recognize that this approach doesn't fully resolve the underlying issue without complicating the vault logic significantly, we decided to leave it simple.**

Medium severity issues

Medium severity issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Insufficient documentation (commented)

While the project's codebase is well-commented, the comments only provide brief descriptions of contract functionalities and do not cover the overall project architecture or contract interactions.

Comprehensive documentation is required to streamline both development and audit processes. It should explicitly explain the purpose and behavior of each contract, their interactions, and the main design choices.

Although the development team provided an additional link explaining the management fee calculation formula, the documentation still lacks clear explanations of how the **BaseVault** and **LidoVault** contracts work.

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

M02. Management fee calculation (commented)

In the `BaseVault.collectManagementFee` function, the management fee is calculated based on the current `totalSupply` and the elapsed time since the last collection. This approach ignores any changes in `totalSupply` during the fee period. As a result, if the `totalSupply` increases significantly right before the fee is collected, the charged fee will be much higher than if it reflected the actual `totalSupply` over time.

On the other hand, if there was little or no `totalSupply` for most of the period, the fee may be unfairly high. This enables potential manipulation and leads to inaccurate fee accrual.

Consider calculating management fees using the time-weighted average `totalSupply` over the fee accrual period, rather than the value at the moment of collection.

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

M03. Potential use of stale Chainlink prices (commented)

The `OracleData.getPrice` function fetches price data using Chainlink's `latestRoundData` but does not verify if the returned price is up-to-date.

If the data is stale, `swap` operations and `totalAssets` evaluation might rely on outdated or invalid price information leading to incorrect assets accounting and incorrect shares calculation.

Consider adding an additional check to ensure the price data is recent.

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

M04. Project's roles (commented)

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

- In the **FeeProvider** contract, the owner can:
  - Change management, deposit, withdrawal and perfomance fees without upper limit on these values;
  - Change tiers data (discount value and minimum amount);
  - Can overwrite the `StakedAmount` information for users;
  - Change `signers`, who are responsible for changing the staked amount.
- In the **BaseVault** contract:
  - The `DEFAULT_ADMIN_ROLE` can pause/unpause the contract and make emergency withdrawal;
  - Also, admin role (`DEFAULT_ADMIN_ROLE` or another one) can upgrade the implementation code;

  Consider adding a `PAUSER` role for `pause` and `unpause` functions to reduce `DEFAULT_ADMIN_ROLE` risks.

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

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

M05. Staked amount can be decreased with previous signatures (commented)

In the current implementation, when a user (from `FeeProvider.signers`) makes a signature and then signs another one with a higher amount, the old signature in the `FeeProvider.setStakedAmount` function can still be used to overwrite the user's staked amount with a lower value by anyone.

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

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Code style (commented)

The **BaseVault** contract inconsistently uses `_msgSender` and `msg.sender`.

For instance, the `updateFeeDiscountDeposit` function uses `msg.sender`, while other functions use `_msgSender`. Consider using a single variant to maintain code style.

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

L02. Missing event emission in setter (commented)

The `BaseVault.__BaseVault_init` function sets the `lastTimeManagementFeeCollected` variable without emitting an event.

Similarly, in the **FeeProvider** contract, the functions `initialize`, `setFees`, and `setManagementFee` update the variables `_depositFee`, `_withdrawalFee`, `_performanceFee`, and `_managementFee` without emitting events.

Consider adding events to improve project transparency and facilitate integration with external services.

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

L03. Missing zero address check for fee recipient (commented)

No zero address check for `feeRecipient` address in the constructor of the **BaseVault** contract. If `feeRecipient` is set to the zero address, deposit, withdrawal, performance, and management fee transfers will revert and lock the corresponding protocol functionality for users.

Consider adding a zero address check for `feeRecipient` in the constructor.

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

L04. No revert in case of invalid condition (commented)

The `BaseVault.deposit` function does not revert when `assets == 0` (it returns 0 instead). Consider reverting in this case to improve developer experience and make integration easier.

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

L05. Unused import (commented)

The **IERC20** interface is not used in the **LidoVault** and **IVault** contracts.

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

L06. Using ERC-7201 (commented)

The **FeeProvider** contract currently manages its own storage layout. Consider adopting the ERC-7201 standard in this contract to prevent storage collisions and simplify upgrades.

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

Notes

N01. Minimal logic for emergency withdrawal (commented)

The `BaseVault.emergencyWithdraw` function does not check the minimum amount of tokens received after calling the `_redeemBaseVault` function, unlike the `redeem` function.

However, since this is an emergency withdrawal, the function should include only the minimal logic necessary to save user funds. In this context, calling `_redeemBaseVault` may be redundant, and withdrawing `wstETH` directly would be more appropriate for emergency situations.

> `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

July 1, 2025