LOPSWAP × PESSIMISTIC

SECURITY ANALYSIS

by Pessimistic

This report is public

August 14, 2025

ABSTRACT

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

DISCLAIMER

SUMMARY

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

The audit showed two related critical issues: Access control #1 and Access control #2. The audit also revealed two issues of medium severity: Transferring fees, Rounding issue leading to loss of liquidity. Moreover, several low-severity issues were found.

After the initial audit, the codebase was updated. All the issues were fixed or commented on.

The overall code quality is good.

GENERAL RECOMMENDATIONS

We recommend fixing the remaining issues. We also recommend implementing CI to run tests, calculate code coverage, and analyze code with linters and security tools.

PROJECT OVERVIEW

Project description

For the audit, we were provided with Lopswap project on a private GitHub repository, commit d9e345549a37b6b5ffc33593164cbc4ff23d9630.

The scope of the audit included everything except:

- **contracts/protocols/MooniswapPool.sol**,
- **contracts/libs/VirtualBalance.sol**.

The documentation for the project included Lopswap White paper.

All 64 tests pass successfully. The code coverage is 75%.

The total LOC of audited sources is 511.

Codebase update

After the initial audit, the codebase was updated. For the recheck, we were provided with the new commit 6018f2307d096a8bbfdceaef95e94c326c438c29.

This update included fixes or comments for all issues. The number of tests increased. All 66 tests passed. The code coverage is 75%.

AUDIT PROCESS

We started the audit on August 4, 2025, and finished on August 11, 2025.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project.

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:

- Tokens cannot be stolen from pools;
- Critical functions and pool initialization have proper access controls;
- AMM pricing formulas are mathematically correct with appropriate rounding;
- Flash loan repayments are properly validated and balances correctly tracked;
- No reentrancy vulnerabilities exist through token callbacks or recursive execution;
- Arithmetic operations are protected against overflow/underflow and division by zero;
- Cross-contract interactions and order routing are secure.

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. We also sent the results to the developers in the text file;
- Audit Agent AI contracts analyzer;
- Savant.Chat AI contracts analyzer.

We ran tests and calculated the code coverage.

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

After the initial audit, we discussed the results with the developers. On August 13 the developers provided us with an updated version of the code. In this update, most of the issues were resolved, and the remaining ones were addressed with comments.

We reviewed the updated codebase and rescanned the project with the following tools:

- Static analyzer Slither;
- Our plugin Slitherin with an extended set of rules.

Afterwards, we updated the statuses of the issues, and inserted developer comments in the respective sections. Finally, we updated the report.

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. **Access control #1** (fixed)

The `preInteraction` function in the **LOPSwap** contract lacks access control and can be called by anyone. While pool initialization uses the original order's makingAmount and takingAmount values (inverted with ~), the function immediately calls `_preUpdateBalances` with attacker-controlled makingAmount and takingAmount parameters. This allows an attacker to manipulate pool balances by subtracting arbitrary makingAmount from the maker asset balance and adding arbitrary `takingAmount` (adjusted for fees) to the taker asset balance. For uninitialized pools, an attacker can trigger initialization and immediately unbalance it with malicious amounts. For already initialized pools, the attacker can directly manipulate balances to create unfavorable exchange rates, enabling subsequent trades at manipulated prices that drain the maker's liquidity.

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

C02. **Access control #2** (fixed)

Related to issue C1, attackers can grief **FlashLoanPool** pool owners by exploiting the unrestricted `preInteraction` function to "borrow" all available liquidity through manipulated balance updates, effectively disabling the pool. This attack is very cheap and can cause issues for the flash loan providers even though no assets are lost.

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

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. **Transferring fees** (fixed)

The `_transferProtocolFee` function in **LOPSwap** attempts to transfer protocol fees from the contract's own balance using `safeTransfer`, but the **FlashLoanPool** contract does not hold any funds - all transfers should come directly from the taker. In the **FlashLoanPool**'s `_transferAssetAndFee` function (line 72), the protocol fee is incorrectly sent from the contract's balance instead of being collected from the taker via `safeTransferFrom`. This will cause transactions to revert when protocol fees are enabled, as the contract has no balance to transfer from, effectively breaking the flash loan functionality whenever protocol fees are non-zero.

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

M02. **Rounding issue leading to loss of liquidity** (fixed)

The **SimpleAMMPool** contract has two rounding issues that favor takers. First, `_getTakingAmount` (line 157) uses standard division that rounds down the required input amount, allowing takers to pay less than required by the AMM curve. This can result in the pool receiving slightly less than the actual post-fee amount. Second, in `_preUpdateBalances` (line 82), `amountWithoutFee` calculation uses `Math.mulDiv` without rounding mode, which rounds down the amount credited to the pool balance and may break AMM invariant in extreme cases.

Both issues drain value from liquidity providers over time. Calculations should use `Math.Rounding.Ceil` to round in the pool's favor.

> `Comment from the developers:`
> In `_preUpdateBalances` we intentionally do not use `Ceil`. This value is the post-fee amount actually owed to the pool. Rounding it up would over-credit the pool (as if the taker deposited more than remained after fees) and desync accounting versus the real transfers.

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

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. Error messages (fixed)

The contracts lack validation that `protocolFee + providerFee < FEE_BASE`. When combined fees reach or exceed 100%, transactions revert with underflow errors in `_getMakingAmount` and `_getTakingAmount` calculations. When fees equal exactly `FEE_BASE`, division by zero occurs. These arithmetic failures produce unclear error messages instead of descriptive reverts, degrading user experience.

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

L02. Usage of mulDiv (fixed)

The contracts inconsistently use `Math.mulDiv` for proportional calculations. While some functions properly use `mulDiv` to prevent overflow and maintain precision, others use standard multiplication and division operators for similar calculations. Although not critical, consistent use of `Math.mulDiv` for all proportional calculations would improve code consistency and eliminate potential precision loss or overflow risks.

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

L03. Setting liquidity on invalidated pool (commented)

The `setLiquidity` function in the **LOPSwap** contract can still be called after `invalidatePool` has been executed. While `invalidatePool` sets the pool status to `INVALIDATED` and prevents further trades through `preInteraction`, the `setLiquidity` function only checks if the pool status is not `NEW` (line 208), allowing liquidity modifications on invalidated pools.

> `Comment from the developers:`
> **Noted. `setLiquidity` doesn't move funds or expose liquidity on `INVALIDATED` pools. Once `INVALIDATED`, trading is blocked and cannot be re-enabled, `setLiquidity` only updates local params with no effect.**

L04. Acceptance of the ether (commented)

The **LOPSwap** contract includes a `receive` function to accept ETH, but the protocol does not actually support ETH operations. Unlike the `OrderMixin` in the 1inch protocol which handles ETH through address(0), **LOPSwap** only works with ERC20 tokens. The `receive` function appears to be included only for the `rescueFunds` functionality, but accepting ETH without proper support could mislead users into sending ETH that becomes stuck in the contract.

> `Comment from the developers:`
> Noted. `receive()` is kept for forward-compatibility with potential native-ETH extensions without changing the base contract. The current release is ERC20-only and does not process ETH.

L05. Pausing the protocol (fixed)

The `paused` check occurs in `postInteraction` (line 341) after `preInteraction` has already been executed by the Limit Order Protocol. This means when paused, `preInteraction` still initializes pools and updates balances via `_preUpdateBalances`, consuming gas and modifying state, before the transaction eventually reverts in `postInteraction`. Moving the pause check to `preInteraction` would fail faster, save gas, and avoid unnecessary state modifications that get reverted anyway.

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

Notes

N01. **Griefing** (commented)

Makers can frontrun order executions to cause them to fail or execute at worse rates by calling `setLiquidity` or `invalidatePool` in the **LOPSwap** contract. Unlike traditional AMMs where liquidity manipulation requires actual token transfers, here it is cheaper as makers can instantly modify pool balances through `setLiquidity` without moving funds. More critically, `setLiquidity` allows changing exchange rates mid-flight, potentially sandwiching takers between unfavorable rates. Takers should use slippage protection.

> The developers added warning comment to the **LOPSwap** contract.

N02. **Order salt layout discrepancy** (commented)

According to the 1inch documentation, the upper 96 bits of `order.salt` should be used for salt. However, the **LOPSwap** contract repurposes several upper bytes of the salt for storing provider fees and flags through the **OrderSaltLib** library.

> `Comment from the developers:`
> Noted. We intentionally pack `providerFee`/flags into salt via **OrderSaltLib**. **LimitOrderProtocol** treats salt as an opaque 256-bit value (signatures and uniqueness are preserved) so this layout doesn't affect **LimitOrderProtocol** behavior.

N03. **Creation of the pools** (commented)

Creating orders in **LOPSwap** costs nothing - makers can create orders without tokens or approval. This enables orderbook pollution with fake orders that force aggregators and interfaces to constantly validate against on-chain state, increasing infrastructure costs and making real liquidity harder to discover.

> `Comment from the developers:`
> Noted. Aggregators can call `LOPSwap.getBalances(...)`, which returns both the pool's accounting balances and balances capped by actually available tokens, so fake/empty orders are trivially filterable. We'll also publish a real-time orderbook feed with up-to-date pool balances to ease discovery for traders, acknowledging the system's dynamic state.

N04. Potential malicious orders (commented)

There are numerous ways to craft malicious orders that will revert on execution, incorrectly shortchange the maker, or attempt to steal from the taker. Orders can contain invalid parameters or malformed extensions that cause unexpected behavior. Both makers and takers must validate order correctness and simulate execution before submission or acceptance, as the protocol itself provides minimal validation.

> `Comment from the developers:`
> Noted. All orders including all params are validated offchain before orderbook admission. Invalid or malicious orders are rejected and do not enter the orderbook.

N05. Complex structure of the orders (commented)

Orders in **LOPSwap** have complex structures requiring specialized tools for creation. Users must correctly set multiple fields including salt-encoded fees, assets, amounts, and extensions. Execution also requires properly formatted `takerInteraction` parameters. Without good tooling, users will struggle to create valid orders or execute trades, making hard to use the protocol directly.

> `Comment from the developers:`
> Noted. Order authoring is tooling-first and aligned with the 1inch LOP ecosystem. Makers use our SDK/UI to build valid orders (including salt-encoded fees); takers don't craft orders manually. Many integrators already work with 1inch **LimitOrderProtocol** and custom extensions, so this is a UX topic rather than a protocol-safety issue.

N06. Non-standard tokens (commented)

The system does not work with non-standard tokens. Tokens with transfer fees, rebasing mechanisms, or any other non-standard behavior tokens are not supported as they will break calculations and balance tracking.

> `Comment from the developers:`
> Noted.

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Yhtyyar Sahatov, Security Engineer
Irina Vikhareva, Project Manager
Alexander Seleznev, CEO

August 14, 2025