

Jupiter Lend DEX

Security Assessment

June 25th, 2026 — Prepared by OtterSec

Gabriel Ottoboni ottoboni@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

Vulnerabilities **5**

OS-JLD-ADV-00 | Fee Charged on Unswapped Amounts 6

OS-JLD-ADV-01 | Split-Leg Rounding Underfunds SwapOut 7

OS-JLD-ADV-02 | Revenue Cut Breaks Input Bounds 8

General Findings **9**

OS-JLD-SUG-00 | Unbounded Shift Duration 10

OS-JLD-SUG-01 | Misleading Comments 11

Appendices

Vulnerability Rating Scale **13**

Procedure **14**

© 2026 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Jupiter engaged OtterSec to assess the `lend-dex` program. This assessment was conducted between May 19th and June 4th, 2026. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a vulnerability where the InOneToken paths overcharge fees by applying the fee to the full single-token amount rather than only the internally swapped leg. This renders them more expensive than an equivalent withdrawal or payback combined with a separate swap (OS-JLD-ADV-00).

Additionally, SwapOut may underfund Liquidity Layer actions because it rounds the combined input once, but rounds ColDeposit and DebtPayback separately. For tiny swaps, the separately rounded legs may sum to more than the input amount, resulting in the downstream Liquidity Layer call to fail (OS-JLD-ADV-01).

Furthermore, high fee and revenue-share settings may result in the amount the DEX actually sends to differ from the amount the Liquidity Layer expects to receive. If that difference is larger than the Liquidity Layer's allowed extra-input tolerance, the Liquidity Layer operation fails (OS-JLD-ADV-02).

We also recommended enforcing logical upper limits to the shift-time and threshold-shift-time to prevent large values (OS-JLD-SUG-00), and further advised ensuring that the code documentation aligns with the implementation logic (OS-JLD-SUG-01).

© 2026 Otter Audits LLC. All Rights Reserved.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Instadapp/fluid-contracts-solana. This audit was performed against c800d0b.

Brief descriptions of the programs are as follows:

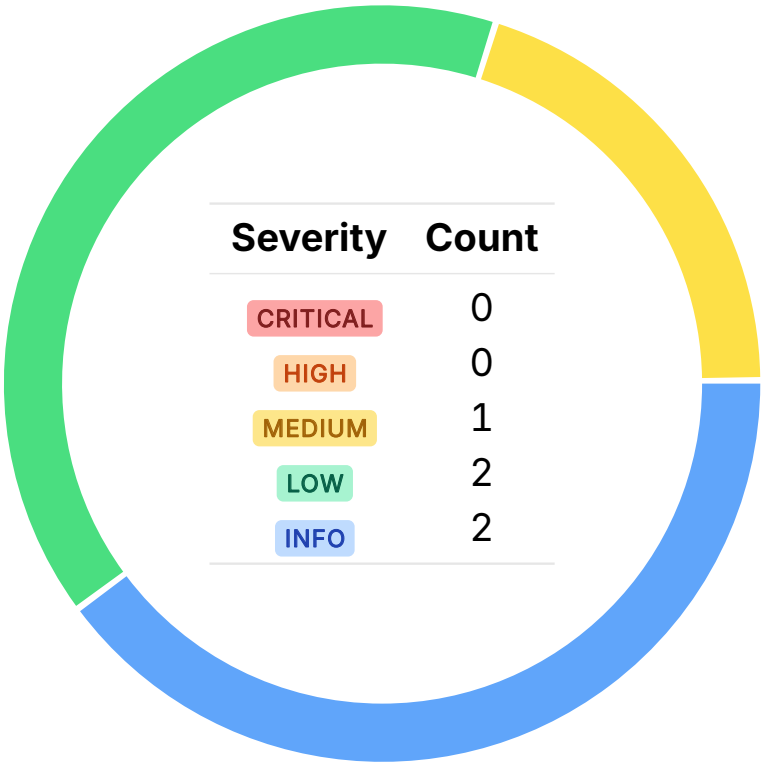
Name	Description
lend-dex	A Solana lending/borrowing protocol that pools liquidity into a shared base layer, letting users supply assets for yield or borrow against collateral.

© 2026 Otter Audits LLC. All Rights Reserved.

03 — Findings

Overall, we reported 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	2
INFO	2

© 2026 Otter Audits LLC. All Rights Reserved.

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-JLD-ADV-00	MEDIUM	RESOLVED	The `in_one_token` paths overcharge fees by applying the fee to the full single-token amount rather than only the internally swapped leg. This renders them more expensive than an equivalent withdrawal or payback combined with a separate swap.
OS-JLD-ADV-01	LOW	RESOLVED	`swap_out` may underfund Liquidity Layer actions because it rounds the combined input once, but rounds `col_deposit` and `debt_payback` separately. For tiny swaps, the separately rounded legs may sum to more than `amount_in`, resulting in the downstream Liquidity Layer call to fail.
OS-JLD-ADV-02	LOW	RESOLVED	High fee / `revenue_cut` settings may make the DEX's actual transferred input diverge from the Liquidity Layer's expected `operate_amount_in`. If that divergence exceeds Liquidity's `MAX_INPUT_AMOUNT_EXCESS` tolerance, the Liquidity Layer `operate` call reverts.

© 2026 Otter Audits LLC. All Rights Reserved.

Fee Charged on Unswapped Amounts MEDIUM OS-JLD-ADV-00

Description

`withdraw_perfect_in_one_token` and `payback_perfect_in_one_token` apply the fee multiplier to the entire single-token amount, including the user's proportional entitlement/repayment portion that did not require a swap. This overcharges users relative to composing `withdraw_perfect` / `payback_perfect` with `swap_in`, where only the swapped leg would incur swap fees, allowing users to pay fewer fees via this path.

Remediation

Apply fees only to the swapped component.

Patch

Resolved in fd1b489.

© 2026 Otter Audits LLC. All Rights Reserved.

Split-Leg Rounding Underfunds SwapOut LOW OS-JLD-ADV-01

Description

`swap::execute_swap_out` rounds the aggregate user input with `descale_up(amt_in_col + amt_in_debt)` but rounds each Liquidity Layer action separately with `descale_up(col_deposit)` and `descale_up(debt_payback)`. For very small swaps, each split leg may round up to one native atom, while the combined `total_in` rounds to fewer atoms than the sum of `col_deposit_native` and `debt_payback_native`. The user module then transfers only `result.amount_in` as the ideal input while asking the Liquidity Layer to process both `col_deposit` and `debt_payback`, creating an internal funding mismatch which results in a panic.

Remediation

Enforce that the sum of `col_deposit_native` and `debt_payback_native` is less than or equal to `amount_in`.

Patch

Jupiter acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Revenue Cut Breaks Input Bounds LOW OS-JLD-ADV-02

Description

High `revenue_cut` may make the DEX request Liquidity Layer accounting amounts that are materially below or above the actual token amount transferred, because `execute_swap_out` derives `col_deposit` / `debt_payback` from fee-adjusted input using `revenue_cut`. Liquidity operate then enforces `net_amount_in` to be within `operate_amount_in` plus only `MAX_INPUT_AMOUNT_EXCESS` tolerance, so aggressive fee/revenue settings may push the transfer/accounting delta outside that bound and cause the Liquidity Layer call to revert.

Remediation

Cap fee/revenue-cut combinations.

Patch

Jupiter acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-JLD-SUG-00	`update_range_percents` and `update_threshold_percent` allow unbounded shift durations, so a large `shift_time` value may leave the DEX stuck in a long active transition.
OS-JLD-SUG-01	There are multiple instances of incorrect comments where the documentation diverges from the actual implementation logic.

© 2026 Otter Audits LLC. All Rights Reserved.

Unbounded Shift Duration OS-JLD-SUG-00

Description

`admin::update_range_percents` and `admin::update_threshold_percent` accept arbitrary nonzero `shift_time` values and only prevent starting a new shift while one is active. If an admin accidentally sets an extremely large duration, the DEX remains locked into a long-running parameter transition with no explicit cancel/reset path.

Remediation

Enforce logical upper limits to `shift_time` and `threshold_shift_time` to prevent large values.

Patch

Resolved in 4293fb4.

© 2026 Otter Audits LLC. All Rights Reserved.

Misleading Comments

OS-JLD-SUG-01

Description

1. The comment in `dex::calc_fee_factors` is misleading because it states that `revenue_cut` is scaled by six decimals. In practice, `revenue_cut` is stored as a compact percentage value scaled by `1e2`, not `1e6`.

>_ programs/dex/src/state/dex.rs RUST

```rust
pub fn calc_fee_factors(&self) -> Result<(u128, u128)> {
    // fee in 4 decimals
    let fee = self.fee as u128;
    // revenueCut in 6 decimals, to have proper precision
    let revenue_cut = self.revenue_cut as u128;
    // 1 - fee. If fee is 1% then withoutFee will be 1e6 - 1e4
    let without_fee = SIX_DECIMALS.safe_sub(fee)?;
    let revenue_cut_factor = EIGHT_DECIMALS.safe_sub(revenue_cut.safe_mul(fee)?)?;
    Ok((without_fee, revenue_cut_factor))
}
```

2. The documentation for `dex::center_price_shift_time` states that the shift continues until the current price reaches the target, or the shift duration ends. However, `center_price_shift_time` is used as a rate denominator, not as a hard end timestamp for the shift. It does not stop merely because a duration has elapsed. It keeps shifting from the last updated center price until the computed shifted value reaches the external target, then resets the shift state.

>_ programs/dex/src/state/dex.rs RUST

```rust
/// This function gradually shifts the center price towards a new target price over time
/// It uses an external price source (via ICenterPrice) to determine the target price
/// The shift continues until the current price reaches the target, or the shift duration
    ↪ ends
/// Once the shift is complete, it updates the state and clears the shift data
/// The shift rate is dynamic and depends on:
/// - Time remaining in the shift duration
/// - The new center price (fetched externally, which may change)
/// - The current (old) center price
/// This results in a fuzzy shifting mechanism where the rate can change as these
    ↪ parameters evolve
/// The externally fetched new center price is expected to not differ significantly from
    ↪ the last externally fetched center price
pub fn calc_center_price_shift(&mut self, new_center_price: u128, now: u128) ->
    ↪ Result<u128> {
    [...]
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

3. The `InitDexParams::center_price` comment is misleading as it says the value uses *"9 decimal"* precision, but the DEX math uses `PRICE_PRECISION` of 10^{15} for price ratios.

```rust
>_ programs/dex/src/state/context.rs
#[derive(AnchorSerialize, AnchorDeserialize, Clone)]
pub struct InitDexParams {
    /// Initial center price (in price precision, 9 decimal normalized ratio)
    pub center_price: u64,
    [...]
}
```

Remediation

Update the comments to align them with the implementation.

© 2026 Otter Audits LLC. All Rights Reserved.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2026 Otter Audits LLC. All Rights Reserved.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2026 Otter Audits LLC. All Rights Reserved.