

# Jupiter DEX Integration

Security Assessment

July 6th, 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-JDI-ADV-00 | Optional Borrow Accounts Skip Arbitrage 6

OS-JDI-ADV-01 | Stale Smart Collateral Limit Check 8

OS-JDI-ADV-02 | Ability to Set Rate Magnifiers when the Corresponding Smart Leg is Enabled 9

OS-JDI-ADV-03 | Missing Center-Price CPI Accounts 10

OS-JDI-ADV-04 | Dex CPI Failure on Different Supply/Borrow Dexes in T4 Vault 11

**General Findings** **12**

OS-JDI-SUG-00 | Dead Code 13

OS-JDI-SUG-01 | Missing Validation Logic 14

## Appendices

**Vulnerability Rating Scale** **15**

**Procedure** **16**


© 2026 Otter Audits LLC. All Rights Reserved.

# 01 — Executive Summary

## Overview

Jupiter engaged OtterSec to assess the `vaults-dex-integration` program. This assessment was conducted between May 8th and June 17th, 2026. For more information on our auditing methodology, refer to Appendix B.

## Key Findings

We produced 7 findings throughout this audit engagement.

In particular, we identified a vulnerability where optional borrow-position accounts may be omitted, causing debt amounts to be read as zero. Because arbitrage execution converts debt-reserve errors into None, the DEX may skip the intended collateral/debt arbitrage even when smart debt is enabled (OS-JDI-ADV-00).

Additionally, the smart-collateral withdrawal-gap check uses the pre-withdraw DexPosition, so in perfect withdrawals it may validate against stale supply shares (OS-JDI-ADV-01).

Furthermore, vault configuration initilization allows non-zero rate magnifiers even when the asset is a DEX share leg, creating phantom interest on smart collateral or smart debt shares (OS-JDI-ADV-02).

We also recommended removing dead code (OS-JDI-SUG-00), and advised incorporating additional validation logic (OS-JDI-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 3daf506.

**Brief descriptions of the programs are as follows:**

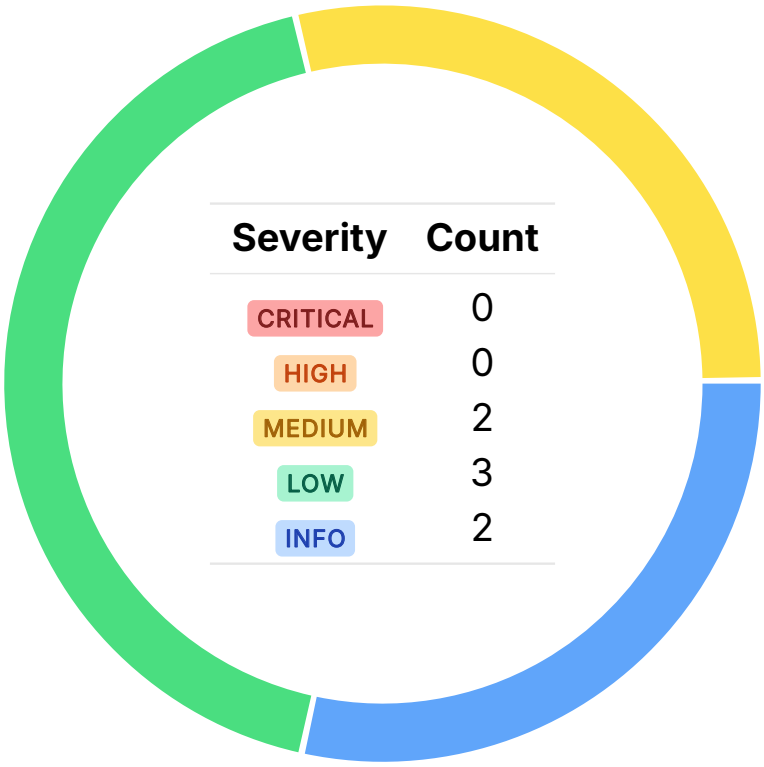
| Name | Description |
| --- | --- |
| vaults-dex-integration | It adds typed vaults whose collateral and/or debt legs can be backed by DEX share positions instead of plain Liquidity Layer token positions. It also includes DEX-backed operate, liquidation, and rebalance paths that reuse vault core accounting, then settle the matching DEX/Liquidity CPIs. |

© 2026 Otter Audits LLC. All Rights Reserved.

# 03 — Findings

Overall, we reported 7 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 | 2 |
| LOW | 3 |
| 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-JDI-ADV-00 | MEDIUM | RESOLVED | Borrow-position accounts are optional, so omitted debt accounts may make debt amounts read as zero. Because `execute_arbitrage` converts debt-reserve errors into `None`, the DEX may skip the intended collateral/debt arbitrage even when smart debt is enabled. |
| OS-JDI-ADV-01 | MEDIUM | RESOLVED | The smart-collateral withdrawal-gap check uses the pre-withdraw `DexPosition`, so in perfect withdrawals it may validate against stale supply shares. |
| OS-JDI-ADV-02 | LOW | RESOLVED | `init_vault_config` allows non-zero rate magnifiers even when the asset is a DEX share leg, creating phantom interest on smart collateral or smart debt shares. |
| OS-JDI-ADV-03 | LOW | RESOLVED | Perfect DEX one-token CPIs do not forward the DEX center-price remaining accounts. If the DEX requires an external center-price account, Vault-to-DEX CPI fails even though the vault-side operation is otherwise valid. |
| OS-JDI-ADV-04 | LOW | RESOLVED | T4 vaults with separate supply and borrow DEXes may cross-wire Liquidity positions from one DEX into a CPI for the other. The DEX validation then rejects them because each position's protocol must match the active `dex_key`. |

© 2026 Otter Audits LLC. All Rights Reserved.

## Optional Borrow Accounts Skip Arbitrage MEDIUM OS-JDI-ADV-00

### Description

In the current implementation, because the DEX borrow-position accounts are optional, omitting them makes `read_borrow_amounts_from_positions` return `(0, false)` for the missing debt leg.

>_ *programs/dex/src/utils/helpers.rs* RUST

```rust
pub fn read_borrow_amounts_from_positions([...]) -> Result<((u128, bool), (u128, bool))> {
    let t0 = if let Some(ref pos) = pos_0 {
        let user_borrow_position = pos.load()?;
        (
            user_borrow_position.get_amount()?,
            user_borrow_position.with_interest(),
        )
    } else {
        (0, false)
    };
    [...]
    Ok((t0, t1))
}
```

When `swap::execute_arbitrage` later calls `dex.get_debt_reserves(...).ok()`, any resulting debt-reserve construction error is swallowed and converted into `None`. The function then returns only a price update with zero arbitrage operations, even if `is_smart_debt` is true. A user may therefore omit required borrow-position accounts to skip the intended collateral/debt rebalance even with both smart legs enabled.

>_ *programs/dex/src/utils/swap.rs* RUST

```rust
#[allow(clippy::too_many_arguments)]
pub fn execute_arbitrage([...]) -> Result<ArbitrageResult> {
    [...]
    let d = if is_smart_debt {
        dex.get_debt_reserves(
            &pex,
            total_amounts.token_0_borrow_amt,
            total_amounts.token_1_borrow_amt,
        )
        .ok()
    } else {
        None
    };
    [...]
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

### Remediation

Require borrow-position accounts whenever smart debt is enabled, and return debt-reserve calculation errors instead of swallowing them with `.ok()`.

### Patch

Resolved in 9686fde.

© 2026 Otter Audits LLC. All Rights Reserved.

## Stale Smart Collateral Limit Check MEDIUM OS-JDI-ADV-01

### Description

In `operate::operate_core`, the smart-collateral withdrawal-gap check within `check_if_smart_col_withdrawal_safe_for_withdrawal_gap` derives the withdrawal limit from the current `DexPosition` before the DEX withdrawal has been applied. For perfect-share withdrawals, the DEX position's withdrawal limit may be recalculated/consumed only when the later DEX operation runs. Thus, because the check uses pre-withdraw supply shares, it will compare the configured gap against a stale position state rather than the post-operation balance. This renders the limit calculation inaccurate for perfect withdrawals, as the enforced limit may not reflect the actual withdrawal being performed.

### Remediation

Compute the withdrawal-gap check against the projected post-withdraw DEX position, or pass the intended share delta into the check explicitly.

### Patch

Resolved in b16e666.

© 2026 Otter Audits LLC. All Rights Reserved.

## Ability to Set Rate Magnifiers when the Corresponding Smart Leg is Enabled

LOW

OS-JDI-ADV-02

### Description

`admin::init_vault_config` derives `is_smart_col` and `is_smart_debt`, but still accepts arbitrary non-zero `supply_rate_magnifier` and `borrow_rate_magnifier` for those smart legs. For smart collateral/debt, the vault asset is a DEX share position, so applying vault-side interest magnification may artificially inflate share accounting rather than reflect real underlying yield, creating phantom interest on DEX shares.

### Remediation

Add checks to ensure that supply and borrow rate magnifiers are set to zero for smart collateral and smart debt legs.

### Patch

Resolved in 9ee89bd.

© 2026 Otter Audits LLC. All Rights Reserved.

## Missing Center-Price CPI Accounts LOW OS-JDI-ADV-03

### Description

`dex::operate_perfect_dex` expects only the perfect-operation remaining-account layout, which omits the DEX center-price remaining accounts used by imperfect CPI paths.

However, `withdraw_perfect_in_one_token_with_signer` and `payback_perfect_in_one_token_with_signer` build plain signed CPI contexts and do not attach any center-price remaining accounts. If the target DEX has `center_price_address` configured, those `in-one-token` instructions may require the external center-price account to compute pricing, so the Vault-to-DEX CPI may fail despite valid vault inputs.

### Remediation

Extend `operate_perfect_dex`, `liquidate_perfect_dex`, and related functions to accept and forward external center-price accounts for `in-one-token` operations.

### Patch

Resolved in b3da9c4.

© 2026 Otter Audits LLC. All Rights Reserved.

## Dex CPI Failure on Different Supply/Borrow Dexes in T4 Vault LOW OS-JDI-ADV-04

### Description

`DexAccount::to_dex_cpi` assumes one DEX CPI may be built by taking the active leg's pool accounts from self while cross-wiring the opposite leg's Liquidity positions from cross. For a T4 vault with distinct supply and borrow DEX pools, this may pass borrow-position accounts whose protocol is the borrow DEX into a CPI executed against the supply DEX, or vice versa. The DEX then rejects the CPI in `validate::validate_liquidity_positions`, because each supplied Liquidity position must have `protocol == dex_key` and the correct mint for the DEX being called.

>_ *programs/dex/src/utils/validate.rs* RUST

```rust
fn validate_liquidity_positions(
    supply_0: &Option<AccountLoader<UserSupplyPosition>>,
    supply_1: &Option<AccountLoader<UserSupplyPosition>>,
    borrow_0: &Option<AccountLoader<UserBorrowPosition>>,
    borrow_1: &Option<AccountLoader<UserBorrowPosition>>,
    dex_key: Pubkey,
    token_0_mint: Pubkey,
    token_1_mint: Pubkey,
) -> Result<()> {
    if let Some(pos) = supply_0 {
        let p = pos.load()?;
        if p.protocol != dex_key || p.mint != token_0_mint {
            return Err(error!(ErrorCodes::DexValidateInvalidLiquidityPosition));
        }
    }
    [...]
}
```

### Remediation

Model T4 as two independent DEX pools at the vault layer, so supply CPIs use only the supply DEX's positions, and borrow CPIs use only the borrow DEX's positions instead of cross-wiring positions across pools.

### Patch

Resolved in 3f32ae8.

© 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-JDI-SUG-00 | The DEX-peg validity check is dead code since failing `!source.is_valid()` already implies failing the DEX peg check. |
| OS-JDI-SUG-01 | Suggestions to ensure incorporation of proper validation logic for improved robustness and to mitigate potential security risks. |

© 2026 Otter Audits LLC. All Rights Reserved.

## Dead Code

OS-JDI-SUG-00

### Description

The `!source.is_valid_dex_peg_source()` check in `oracle::init_oracle_config` is redundant if every DEX-peg-valid source already satisfies `source.is_valid()`. In that case, once `!source.is_valid()` is true, the DEX-peg validity check cannot rescue the source, so the condition is equivalent to only checking `!source.is_valid()`

>_ *programs/oracle/src/lib.rs* RUST

```rust
pub fn init_oracle_config(
    ctx: Context<InitOracleConfig>,
    sources: Vec<Sources>,
    nonce: u16,
) -> Result<()> {
    [...]
    let mut i = 0;
    while i < sources.len() {
        let source = &sources[i];
        if !source.is_valid() && !source.is_valid_dex_peg_source() {
            return err!(ErrorCodes::InvalidParams);
        }
        [...]
    }
    [...]
}
```

### Remediation

Simplify the condition to `!source.is_valid()`.

© 2026 Otter Audits LLC. All Rights Reserved.

## Missing Validation Logic OS-JDI-SUG-01

### Description

1. Currently, `preview::preview_dex_shares` reuses real DEX execution math but does not run the same account-context validation as state-changing DEX operations. If callers pass mismatched reserves, positions, token mints, or oracle accounts, the preview may return misleading share deltas even though the equivalent real operation would reject. Add the same validation checks used by executable DEX paths.
2. `oracle::init_dex_peg_oracle_config` stores DEX, position, and reserve account keys after validation, so it should ensure those accounts are actually owned by the expected DEX/Liquidity programs and have the expected account discriminator/type.

### Remediation

Incorporate the above validations.

© 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.