Jupiter Lending

Security Assessment

November 25th, 2025 — Prepared by OtterSec

Gabriel Ottoboni ottoboni@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Jupiter engaged OtterSec to assess the `lending` and `lendingRewardRateModels` programs. This assessment was conducted between November 12th and November 20th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

In particular, we recommended removing the unnecessary claim account from the Withdraw context (OS-JLE-SUG-00). We further advised validating the supply token reserves liquidity account in the Withdraw and Deposit account for improved robustness (OS-JLE-SUG-01).

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 1c0bba0.

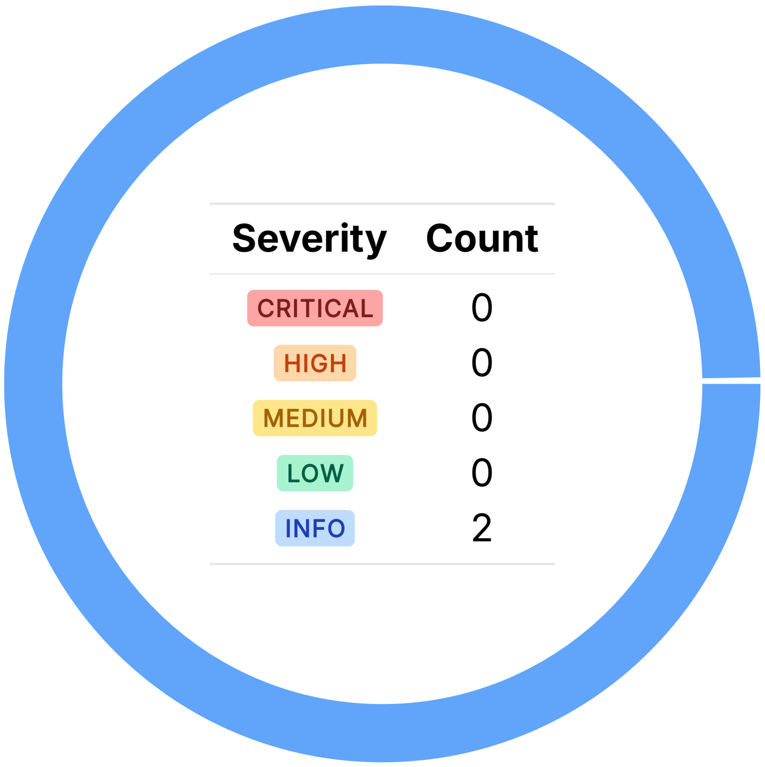
A brief description of the program is as follows:

Name	Description
lending	An implementation of a Solana-based lending protocol, allowing users to deposit, withdraw, mint, and redeem tokens while earning interest.
lendingRewardRate-Models	It manages reward distribution for the lending protocol, including starting, stopping, and scheduling reward cycles.

03 — Findings

Overall, we reported 2 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	0
LOW	0
INFO	2

04 — 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-JLE-SUG-00	`claim_account` is unnecessary in `Withdraw` because all withdrawals utilize `TransferType::DIRECT`.
OS-JLE-SUG-01	The lending program trusts an unchecked `supply_token_reserves_liquidity` account and relies solely on a CPI for validation, which becomes unsafe if any future codepath utilizes this account without calling the CPI.

Redundant Claim Account

OS-JLE-SUG-00

Description

In the lending program, `user::withdraw` currently requires a `claim_account`, which is unnecessary, since the withdrawal is always executed with the `TransferType::DIRECT` option.

>_ *programs/lending/src/state/context.rs* RUST

```rust
#[derive(Accounts)]
pub struct Withdraw<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    [...]

    #[account(mut)]
    /// CHECK: Safe as this will be verified in liquidity program CPI call
    pub claim_account: UncheckedAccount<'info>,
    [...]
}
```

Remediation

Change the CPI to utilize `None` and remove or make the account optional.

Patch

Resolved in 4f557be.

Missing Direct Reserve Validation OS-JLE-SUG-01

Description

In the lending program, the `Withdraw` and `Deposit` account structures do not validate the `supply_token_reserves_liquidity` account, instead relying on a CPI to the liquidity program to do so. Since this account's utilization is not limited to the CPI, this may result in issues if some codepath that does not perform the CPI is ever added to the program.

>_ *programs/lending/src/state/context.rs* RUST

```rust
#[derive(Accounts)]
pub struct Withdraw<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,
    [...]

    // Here we will list all liquidity accounts, needed for CPI call to liquidity program
    // @dev no verification of accounts here, as they are already handled in liquidity program,
    // Hence loading them as UncheckedAccount
    #[account(mut)]
    /// CHECK: Safe as this will be verified in liquidity program CPI call
    pub supply_token_reserves_liquidity: AccountLoader<'info, TokenReserve>,
    [...]
}
```

Remediation

Validate this account in the lending program so that the CPI is not a prerequisite for the validation to occur.

Patch

Resolved in 4b5c2f8.

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.

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.